UNIVERSIDAD CARLOS III DE MADRID

DEPT. TEORÍA DE LA SEÑAL Y COMUNICACIONES

Proyecto Fin de Carrera

DISEÑO Y DESARROLLO DE UN JUEGO PARA PC Y ANDROID PARA EL APRENDIZAJE DE LA SUMA Y LA MULTIPLICACIÓN (ESCOBA²)

INGENIERÍA DE TELECOMUNICACIÓN

Autor: Xavier Verguín González
Tutor: Dr. Víctor P. Gil Jiménez

Leganés, Octubre 2014

Agradecimientos

Llegar al final de esta etapa de mi vida, que se cierra con el proyecto que aquí se presenta, no habría sido posible sin la ayuda y el apoyo de mucha gente. Por ello me gustaría compartir y dedicar a todos vosotros este momento.

A Víctor, mi tutor, por su ayuda durante este último año y por la confianza que ha depositado en mí para realizar este proyecto, que aúna muchas de las competencias que he adquirido durante estos años de ingeniería. Gracias, también, por haberme brindado la oportunidad de realizar un trabajo dirigido bajo tu dirección.

A todos los profesores que durante todos estos años – de universidad, instituto y colegio - han aportado su granito de arena y me han ayudado para que un día como hoy pueda estar presentando este proyecto fin de carrera.

A mi madre, porque sin su sacrificio y apoyo incondicional no estaría escribiendo estas líneas en este momento. A mi padre, por transmitirme su pasión por la ciencia y la ingeniería en cada momento desde que era un niño. A mi hermana, por el gran ejemplo y fuente de inspiración que has sido durante todos estos años.

A toda mi familia, por haberme ayudado a ser quien soy desde que no era más que un niño. Especial mención a mis tíos Gilles y Mª José, por compartir y ayudarme durante tantas navidades previas a mis épocas de exámenes. Gracias también a mi tía Cristina por acompañarme en Leganés durante estos años de ingeniería.

A mis compañeros de residencia y ahora inseparables amigos: Antonio, José, Dani, Quique, Mikel y Enrique por estar compartiendo todos los momentos desde que empezó esta aventura en septiembre del 2008. Gracias también a mis compañeros de carrera con los que tantos buenos momentos, horas de prácticas y exámenes he pasado: Dani (¡estás en todos lados eh!), Pablo, Irene, Lucía, Aránzazu, Yolanda y toda la promoción. A Pedro, por la gran ayuda que me ha brindado durante estos últimos meses.

Y, especialmente, a Joana. Gracias por ser como eres, por estar a mi lado, por aguantarme durante todos estos años y por darme la oportunidad de compartir contigo todos y cada uno los momentos más bonitos de mi vida desde 2010.

A todos vosotros, gracias.

Xavier

Resumen

Diseño teórico e implementación del juego de cartas ESCOBA² para las plataformas Microsoft Windows y Android.

Este juego educativo tiene como finalidad el aprendizaje, afianzamiento y entrenamiento de las operaciones básicas como la suma y la multiplicación, a la vez que constituye un entretenimiento y diversión para los públicos de todas las edades.

El presente proyecto fin de carrera aborda tanto el diseño teórico e implementación de los distintos algoritmos de inteligencia artificial utilizados por los jugadores máquina en los distintos modos de juego de la ESCOBA², como el diseño e implementación de la interfaz de usuario y de todas las funcionalidades adicionales propias de un juego y necesarias para este proyecto.

Abstract

Design and implementation of the card game ESCOBA² for Microsoft Windows and Android platforms.

This educational game is aimed at learning, strengthening and training of basic operations such as addition and multiplication, while constituting an entertainment and fun for audiences of all ages.

This master thesis addresses both, the theoretical design and the implementation of different artificial intelligence algorithms used by machine players in the different game modes of the ESCOBA², and the design and implementation of the user interface and all the additional features required by a game and thus needed for this project.

Índice

Índice de figuras

Índice de tablas

1 Introducción

El presente proyecto fin de carrera tiene como objetivo el diseño teórico e implementación del juego de cartas ESCOBA² para las plataformas *Microsoft Windows* [1] y *Android* [2].

La ESCOBA² es un nuevo juego de cartas que presenta unas reglas derivadas del tradicional juego *la escoba* [3], las cuales extiende, dotándoles de una mayor profundidad e incluyendo además la operación de multiplicación al sistema de puntuación.

Este proyecto fin de carrera se realizará en cinco fases:

1. En primer lugar, se evaluarán las distintas alternativas para implementar el juego ESCOBA², tanto desde un punto de vista algorítmico (*Inteligencia Artificial*) como desde un punto de vista de implementación (conjunto de tecnologías que se utilizarán para su realización).
2. Seguidamente, se seleccionará, para ambos aspectos – *algorítmico e implementación* -, la solución que mejor cumpla con las especificaciones y restricciones impuestas.
3. A continuación, se diseñarán los distintos estados del juego necesarios para implementar la funcionalidad requerida.
4. Tras la fase de diseño, se abordará la implementación del juego tanto para la plataforma de ordenador personal *Microsoft Windows* [1] como para la plataforma móvil *Android* [2].
5. Finalmente, se incluirá un último capítulo de conclusiones y futuras líneas de trabajo. Además, en este último capítulo también se incluirá la planificación del proyecto y el presupuesto.

La estructura de la presente memoria respetará, también, esta misma división en cinco fases para organizar los distintos contenidos generados durante todo el desarrollo del proyecto.

Por tanto, las primeras secciones de la memoria – *de la 1 a la 4* – abordarán tanto el diseño teórico de la parte algorítmica como el estudio y selección de la tecnología de implementación, correspondientes a las dos primeras fases descritas anteriormente.

La tercera fase se tratará en la sección 5, donde nos centraremos en el diseño de los distintos estados y pantallas de juego requeridos para implementar la ESCOBA².

Seguidamente, la cuarta fase abarcará de la sección 6 a la 8 y será donde se estudie la implementación de la aplicación; que abarca desde la descripción del entorno de desarrollo utilizado en las distintas plataformas soportadas hasta el estudio de las optimizaciones realizadas para obtener el excelente rendimiento requerido por este tipo de aplicaciones móviles.

Para finalizar, la quinta y última fase se trata en la última sección de esta memoria – *la sección 9* -, e incluirá las conclusiones finales, planificación del proyecto, presupuesto y las futuras líneas de trabajo, como hemos mencionado anteriormente.

A continuación, proseguiremos esta introducción enumerando y estudiando las reglas de juego de la ESCOBA² y estudiando la dinámica de juego del mismo. Además, también se realizará una introducción tanto de los requisitos funcionales como de los requisitos técnicos necesarios para la implementación del presente proyecto fin de carrera.

1.1 Reglas del juego ESCOBA²

El juego de cartas ESCOBA² es una variante del tradicional juego *la escoba* [3] que se juega con la baraja española. A continuación se detallan tanto las premisas de partida del juego como la dinámica del mismo.

1.1.1 Premisas del juego

- De 2 a 4 jugadores
- Baraja española (40 cartas)
- En cada partida, los jugadores van acumulando puntos y gana quien primero llegue a 30 puntos:
    - Cada escoba: 1 punto
    - El que mayor número de cartas tiene: 1 punto
    - El que mayor número de oros tiene: 1 punto
    - El que mayor número de sietes tiene: 1 punto
    - El que gana el 7 de oros: 1 punto
    - El que gana el 5 de oros: 1 punto
- Numeración de cada carta: as: 1, 2, 3, 4, 5, 6, 7, sota: 8, caballo: 9 y rey: 10.

1.1.2 Dinámica de juego

Inicialmente se reparten 3 cartas a cada jugador y se colocan 4 cartas sobre la mesa, o el número de cartas que haga que en el último reparte le correspondan 3 cartas a cada jugador – recordemos que en cada partida se reparten cartas un número de veces igual a $(40 - \boldsymbol{X})/(NumJugadores \cdot 3)$, por lo que habrá que calcular $\boldsymbol{X}$ para que el resultado de la operación sea un número entero, de forma que en el último reparte se den 3 cartas a cada jugador. Para el caso que nos ocupa (2, 3 y 4 jugadores) seleccionamos $\boldsymbol{X = 4}$.

Por turnos, cada jugador intentará llevarse el mayor número de cartas (o, mejor dicho, aquella combinación de cartas que le proporcione el mayor número de puntos, de acuerdo a la lista anterior). Para ello, tomando una y solo una de sus cartas, deberá multiplicar su valor numérico por cualquiera de las de la mesa y luego podrá sumar al resultado de la multiplicación cuantas cartas sobre la mesa desee, con el objetivo de conseguir un múltiplo de 5 no divisible entre 2: 5, 15, 25, 35,… para conseguir hacer baza y llevarse las cartas que se hayan usado.

Si no puede obtener ninguno de estos números con ninguna combinación, deberá colocar una de sus cartas sobre la mesa y pasará el turno al siguiente jugador.

Si un jugador suma los números anteriores utilizando todas las cartas del tablero, se dice que ha hecho una **escoba**, marcando este evento en su taco de cartas colocando una de ellas boca arriba.

Una vez se han repartido todas las cartas, el último jugador que ha hecho baza se lleva las cartas que todavía queden sobre el tablero.

1.1.3 Variación: asociación

Además de la dinámica de juego anteriormente descrita, la aplicación final contará con la posibilidad de jugar a una variación más avanzada: asociación.

En este modo de juego, cada jugador podrá multiplicar una de sus cartas por otra de las de la mesa o – *ésta es la novedad* - la suma de otras del tablero y, posteriormente, sumar cuantas cartas sobre el tablero se quiera.

1.2 Requisitos

La implementación del juego ESCOBA² está sujeta a una serie de requisitos tanto funcionales como técnicos. A continuación se detallan de forma clara y concisa los mismos.

1.2.1 Funcionales

Los requisitos funcionales son aquellos que definen el comportamiento que ha de tener la aplicación final, sin imponer restricciones sobre cómo o con qué tecnologías se implementa.

- Número de jugadores: 2, 3 o 4. Un jugador será el usuario mientras que el resto de jugadores serán I.A. proporcionada por el juego.
- Dos modos de juego: estándar y asociación.
- Modo torneo: gana el mejor de 2, 3, 4,… o 20 rondas. Número de rondas especificado por el usuario.
- Distintos niveles de dificultad: fácil, medio y difícil.
- Generación de estadísticas: tanto de los juegos simples como de las distintas rondas y del torneo completo en modo torneo.

1.2.2 Técnicos

A continuación se muestras los requisitos de índole técnica, es decir, aquellos que añaden restricciones a la implementación.

- Multiplataforma: *Microsoft Windows* [1] y *Android* [2].
- Tamaño reducido: menor de $10\ MB$.
- Uso eficiente de los recursos del sistema: memoria y procesador.
- Cálculo de estrategia de juego en tiempo real, sin esperas notables para el usuario. Retardo menor a $500ms$.
- Resolución mínima soportada: $240\ x\ 320$ píxeles.
- Mínima versión de Android soportada: 2.1.
- No ha de requerir acceso a la red (aplica a Android).
- Mínimo set de instrucciones necesario: *ARMv7* [4].
- Compatible con *OpenGL ES 2.0* [5].

2 Inteligencia Artificial

La *Inteligencia Artificial* es la piedra angular para el correcto funcionamiento de este proyecto: tanto la calidad del jugador máquina como los tiempos de espera y consumo de recursos dependerán del correcto diseño e implementación de los algoritmos necesarios.

Como veremos más adelante, la particularización de los algoritmos de búsqueda de soluciones para nuestro caso concreto tendrá que tener en cuenta muchas situaciones debido a la gran combinatoria de estrategias de juego que se dan en nuestro problema: en función del número de jugadores, el modo de juego, los distintos niveles de dificultad, etc.

2.1 Introducción

En primer lugar vamos a ver una pequeña introducción de forma teórica a las técnicas de resolución de problemas y a la nomenclatura que usaremos [6].

Desde un punto de vista teórico, la resolución de problemas es una búsqueda en un espacio de estados, siendo:

$$Estado = < Q, R, C >$$

- Q: ***estructura de datos*** que describe al estado.
- R: ***reglas*** u operaciones que describen las transiciones en el espacio de estados.
- C: estrategia de ***control***.

De tal forma que encontrar la solución consiste en encontrar una secuencia de reglas $r_1, \dots, r_n$ que conduzcan desde el estado inicial q_0 al estado final q_f.

Por tanto, para diseñar nuestro algoritmo de resolución seguiremos los siguientes pasos:

1. Definir un espacio (conjunto) de estados.
2. Especificar uno o más estados iniciales.
3. Especificar uno o más estados finales (meta/objetivo).
4. Definir reglas sobre las acciones disponibles (abstracción del mundo real a un modelo simbólico).
5. El problema se resuelve usando las reglas en combinación con una estrategia de control.
6. La estrategia de control establece el orden de aplicación de las reglas y resuelve conflictos.

Para el modelado teórico y posterior implementación práctica utilizaremos grafos de expansión de estados: representados en este caso por árboles (grafos acíclicos), donde:

- Nodos: estados intermedios
- Arcos: aplicación de un operador

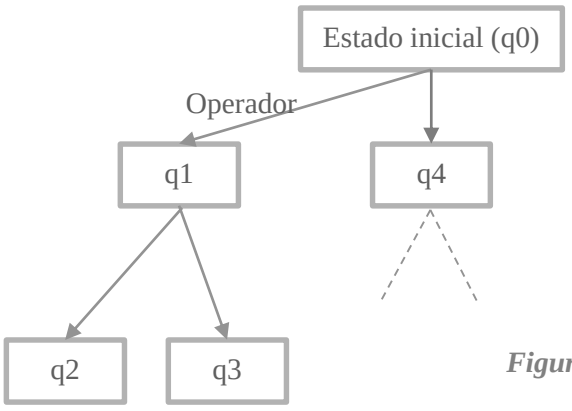


Figura 1 – Árbol de expansión de estados

El estado final (y, por tanto, reglas que conducen a la solución) se seleccionará en función de una serie de criterios que maximicen el beneficio del jugador. Para nuestro caso, estos criterios de maximización del beneficio son razonables, como veremos posteriormente.

2.2 Posibles soluciones

En primer lugar, es importante destacar que la estrategia a seguir cuando tan solo hay un contrincante es distinta de la estrategia que se sigue cuando hay varios. En el primer caso, podemos estimar qué cartas tiene el contrincante en función de las cartas que ya han sido descubiertas, y esto nos da una ventaja sobre el adversario ya que podemos simular todas las posibles situaciones hasta que se terminen las cartas de ambos jugadores y realizar *mini-max* para realizar el movimiento más ventajoso, además de minimizar sus posibilidades cuando no podemos hacer baza y hemos de depositar una carta sobre la mesa.

Si hay varios jugadores, no podemos saber realmente qué cartas va a tener el siguiente jugador (ni realizar una correcta estimación), por lo que en este caso la estrategia de '*contar cartas*' no aporta ventajas significativas y no se implementará.

A continuación se enumerarán las distintas alternativas por las que podemos optar para solucionar nuestro problema:

1. Implementar toda la lógica en un único conjunto de reglas y estrategia de control.
2. Utilizar algoritmos distintos en función del número de jugadores (2 vs 3 o 4).
3. Utilizar algoritmos distintos en función del número de jugadores y modo de juego (estándar/asociación).
4. Utilizar algoritmos distintos en función del número de jugadores, modo de juego y el nivel de dificultad.

En el primero tipo de solución *(1)* contamos con la gran ventaja de solo tener que implementar un algoritmo de evaluación, sin embargo añade complejidad al conjunto de reglas y sistema de control además de la consecuente pérdida de rendimiento debido al aumento de la complejidad de control.

El resto de soluciones proponen la implementación de varios algoritmos (granularidad más fina) con el objetivo de simplificar el conjunto de reglas y el sistema de control. El problema es la necesidad de implementar varios algoritmos y no únicamente uno.

2.3 Selección de la solución

Para seleccionar la solución tendremos en cuenta tanto la complejidad de la implementación (no es rentable implementar decenas de algoritmos distintos) como el rendimiento esperado en una implementación *software*.

2.3.1 Tabla comparativa

A continuación se clasifican en una tabla los distintos tipos de solución en función de las siguientes magnitudes:

- **Complejidad algorítmica**: número de implementaciones distintas de algoritmos de búsqueda de solución que se han de llevar a cabo.
- **Rendimiento**: velocidad en la ejecución del algoritmo. De manera cualitativa se evalúa la pérdida de rendimiento debido a '*cache misses*' [7] (acceso a memoria que se encuentre fuera de la línea de caché actual) y errores en '*branch predictions*' (predicción de saltos) [8] que habrá como consecuencia del gran número de instrucciones condicionales dentro de las reglas.
- **Facilidad de implementación**: facilidad a la hora de realizar la implementación software. Se tiene en cuenta tanto el tamaño del código (implementar múltiples algoritmos requiere reescribir código similar varias veces) como la complejidad de las reglas y del sistema de control.

Tipo de solución *(*)*	**Complejidad algorítmica**	**Rendimiento**	**Facilidad de implementación**
1	1	Bajo	Media
2	2	Medio	Fácil
3	4	Medio	Fácil
4	12	Alto	Difícil

(*) Se hace referencia a los 4 tipos de soluciones propuestos en al apartado 2.2.

Tabla 1 – Tabla comparativa de las distintas soluciones algorítmicas

2.3.2 Selección

Como se puede deducir de la tabla anterior, las opciones 1 y 4 quedan descartadas por encontrarse en los extremos.

La primera de ellas *(1)* tiene la gran ventaja de solo tener que implementar un algoritmo para la búsqueda de la solución, sin embargo su rendimiento es demasiado bajo debido a la gran complejidad de las reglas (que se evalúan en las transiciones entre estados: muchas veces) y la implementación de las reglas y del sistema de control puede llegar a ser compleja (muchos saltos condicionales para adecuarse a los distintos modos de juego, número de jugadores, etc…).

La número cuatro *(4)* puede proporcionar un rendimiento muy alto ya que las reglas y el sistema de control son bastante sencillos, pero por el contrario necesitamos implementar 12 algoritmos distintos: uno para cada combinación posible de número de jugadores (2 opciones: 2 jugadores vs 3 o 4), modo de juego (2 opciones: estándar y asociación) y dificultad (3 opciones: fácil, intermedia y difícil), esto es, 12 combinaciones en total.

Por tanto, nuestra decisión ahora se reduce a evaluar los tipos de solución 2 y 3.

La opción 2 propone implementar dos algoritmos distintos, uno para el modo de 2 jugadores (1 vs 1) y otro para el modo de 3 o 4 jugadores. El resto de comportamientos se implementan en las reglas y el sistema de control.

La opción 3 es similar a la anterior pero implica implementar cuatro algoritmos distintos, ya que además de por el número de jugadores aquí también se diferencia el modo de juego: estándar o asociación.

Debido a que la diferencia en comportamiento de los modos de juego estándar y asociación se puede implementar de forma sencilla dentro de una misma regla, **se ha optado por la segunda solución**: implementar dos algoritmos distintos en función del número de jugadores de la partida e implementar el resto de comportamientos en las propias reglas y sistema de control.

2.4 Estudio detallado de la solución seleccionada

La solución seleccionada implica la implementación de dos algoritmos de búsqueda de solución distintos (aunque con similitudes):

- **2 jugadores**: además de la maximización de la puntuación, se utilizará el conteo de cartas para mejorar la decisión en dos casos:
    1. Cuando hay varias alternativas para hacer baza, en lugar de maximizar la puntuación de esa baza trataremos de maximizar la puntuación total de las siguientes bazas hasta el siguiente reparte. Excepto en un caso:
        - El contrario acaba de dejar una carta sobre la mesa. En este caso se buscará hacer baza con la carta que ha dejado el contrincante (y otras cartas que haya sobre la mesa), ya que eso significa que en el siguiente turno nuestro contrincante tampoco va a poder realizar ningún movimiento y se va a ver obligado a dejar otra carta sobre la mesa. Si hay varias posibles jugadas que involucran a la carta que ha dejado el contrincante, se seleccionará aquella que aporte un mayor valor añadido.
    2. Cuando no sea posible hacer baza (sumar 5, 15, 25,…) y haya que depositar una carta sobre la mesa.
- **3 o 4 jugadores**: se llevará a cabo una estrategia más simple que en el caso de 2 jugadores, buscando únicamente maximizar el número de puntos que obtenemos para las cartas que tenemos actualmente y las que hay sobre la mesa, sin recurrir al conteo de cartas para mejorar la estrategia (no aporta ventajas significativas cuando existe más de un contrincante). También se minimizará la pérdida de beneficio al depositar una carta sobre la mesa, aunque sin apoyarnos en el conteo de cartas como en el caso de 2 jugadores.

2.4.1 Estructuras de datos

En primer lugar, definiremos las estructuras de datos que utilizarán los algoritmos con el fin de establecer una relación directa entre el análisis teórico y la implementación práctica.

Para la aplicación de los algoritmos, distinguimos 3 tipos de estructuras de datos:

1. **Carta**: estructura de datos que define una carta de la baraja.
2. **Lista de cartas**: como su propio nombre indica, es un conjunto de cartas no repetidas, que serán utilizadas para definir los distintos conjuntos de cartas en cada estado: cartas de la mesa, cartas propias, cartas que han salido (conteo de cartas), etc.
3. **Árbol**: estructura de datos utilizada para expandir los estados de búsqueda. Cada estado es un nodo del árbol. Estos estados estarán definidos por distintas listas de cartas e información sobre la puntuación obtenida en cada uno de ellos, como estudiaremos en detalle posteriormente.

2.4.1.1 Carta

La estructura 'Carta' define una carta de la baraja, con las siguientes propiedades:

- **Número**: del 1 al 10.
- **Palo**: oros, bastos, copas o espadas.
- **Valor añadido**: valor numérico que define el valor añadido de la carta. Será utilizado para calcular la mejor combinación a elegir (mayor valor añadido) en caso de que haya múltiples combinaciones de cartas que sumen 5, 15, 25,…

  Si quien posea la carta obtiene un punto más, se sumará 1 (5 y 7 de oros); si por el contrario esta carta pertenece a un grupo de cartas que puede dar un punto más en caso de que el jugador tenga más de la mitad de cartas de este grupo (oros y sietes), se sumará 0.5. El resto de cincos son un caso especial, pues con que haya una carta impar sobre la mesa permiten hacer una jugada válida, por eso su valor añadido será de 0.25. Para el resto de cartas, se le da mayor prioridad a las impares, con el objetivo de impedir que los oponentes puedan realizar una jugada (ya que es necesaria una carta impar sobre la mesa) así como para descartar las cartas pares en lugar de las impares cuando sea posible.
  Para cada carta, este campo tomará el valor:
  - 7 de oros: 1.0 + 0.5 + 0.5 = 2.0
  - 5 de oros: 1.0 + 0.5 = 1.5
  - Resto de sietes: 0.5
  - Resto de oros: 0.5
  - Resto de cincos: 0.25
  - Resto de cartas:
    - 0.01 pares
    - 0.011 impares

2.4.1.2 Lista de cartas

Esta estructura de datos es, simplemente, un conjunto de cartas no repetidas que se utilizará para representar las cartas que hay sobre la mesa, las que posee un jugador, etc.

2.4.1.3 Árbol

Para expandir los distintos estados en busca de una solución utilizaremos una estructura en árbol. Cada nodo define un posible estado y tiene asignada una puntuación. Finalmente, una vez que se hayan expandido todos los posibles estados, se seleccionará aquella solución que sea correcta (5, 15, 25,…) y que además aporte un mayor valor añadido (en caso de existir múltiples soluciones válidas).

Tanto los datos que se almacenarán en los nodos como las reglas y sistema de control que rigen la expansión dependerán de si estamos utilizando conteo (2 jugadores) o no (3 o 4 jugadores), y serán definidos para cada caso posteriormente, no obstante esta es la descripción general:

- **Nodo**: define un estado de la expansión, esto es, el resultado de realizar esa jugada (válida o no, esto se filtrará a posteriori). Para seleccionar la jugada más ventajosa se utilizará un sistema de pesos numérico, función del número de cartas que se obtienen de la jugada, qué cartas son, si la jugada hace escoba o no, etc., denominado '*valor añadido*' y que se calcula como la suma de los valores añadidos de todas las cartas que se obtienen con esa jugada más un punto extra de valor añadido si la jugada hace *escoba* (se utilizan todas las cartas que haya sobre la mesa).
- **Arcos**: el sistema de control aplicará una serie de reglas sobre el nodo padre (estado original) y cada posible combinación generará un estado hijo.

2.4.2 Algoritmo 3 o 4 jugadores: sin conteo

En primer lugar realizaremos un estudio detallado del caso de 3 o 4 jugadores, ya que es más sencillo, y posteriormente se analizarán las partes que sufren variaciones para el caso de 2 jugadores debido al conteo de cartas.

La versión del algoritmo de búsqueda de solución para más de 2 jugadores no tiene en cuenta estrategias basadas en conteo de cartas y la decisión únicamente está basada en la maximización de la puntuación utilizando las cartas que hay sobre la mesa y las del jugador.

A continuación, estudiaremos la expansión del árbol y la posterior elección de la solución, para ello dividiremos el análisis en 3 partes: descripción del estado original (nodo padre), descripción del sistema de control (expansión del árbol) y, finalmente, cómo se realiza la elección de la solución.

2.4.2.1 Nodo padre (estado original)

El algoritmo de búsqueda de solución se aplica sobre un determinado estado de la partida, representado por el denominado nodo padre. Teniendo en cuenta que en esta versión no se utiliza conteo de cartas, el nodo padre debe contener:

- Cartas que posee el jugador para jugar.
- Cartas que hay sobre la mesa.
- Cartas que tiene el jugador en su poder (almacenadas), con el objetivo de hacer una mejor elección cuando el jugador ya tiene más de la mitad oros o sietes.

Para ello se utilizarán tres listas de cartas.

2.4.2.2 Sistema de control (expansión del árbol)

El sistema de control es el encargado de expandir el nodo padre en las posibles jugadas. Para ello, tendrá que aplicar una serie de reglas de expansión en función del tipo de juego (estándar o asociación).

Modo estándar:

Recordemos que, en el modo estándar, el objetivo es encontrar una combinación de cartas tal que multiplicando una de las del jugador por una de las de la mesa y (opcionalmente) sumando tantas cartas como se quiera, se obtenga un múltiplo de 5 no divisible entre 2 (5, 15, 25,…).

Algoritmo:

1. Para cada carta del jugador (j)
    a. Para cada carta de la mesa (m)
        i. Se multiplica j por m
        ii. Se almacena el valor resultante (j*m)
        iii. Se almacenan las cartas involucradas (2 en esta etapa) en una lista.
        iv. Se calcula el valor añadido de la jugada (tiene en cuenta si hay o no 5 o 7 de oros, etc.), sumando el valor añadido de las cartas involucradas.
        v. Si *m* es la última carta que queda sobre la mesa, se añade un punto al valor añadido acumulado de la jugada (pues se hace escoba).
            1. Para cada carta restante de la mesa (m’)
                a. Se suma el valor m’ al resultado anterior: j*m + m’
                b. Se almacena el valor resultante
                c. Se calcula el valor añadido de la jugada
                d. Para cada carta restante sobre la mesa (m’’), se vuelve a realizar el paso anterior, iterativamente, hasta que no haya cartas sobre la mesa.

Llegados a este punto, tenemos el árbol expandido en todas las posibles combinaciones: por tanto, esto incluye jugadas válidas (el valor resultante es 5, 15, 25,…) y jugadas no válidas (el valor resultante no es un múltiplo de 5 no divisible entre 2).

Diagrama del algoritmo:

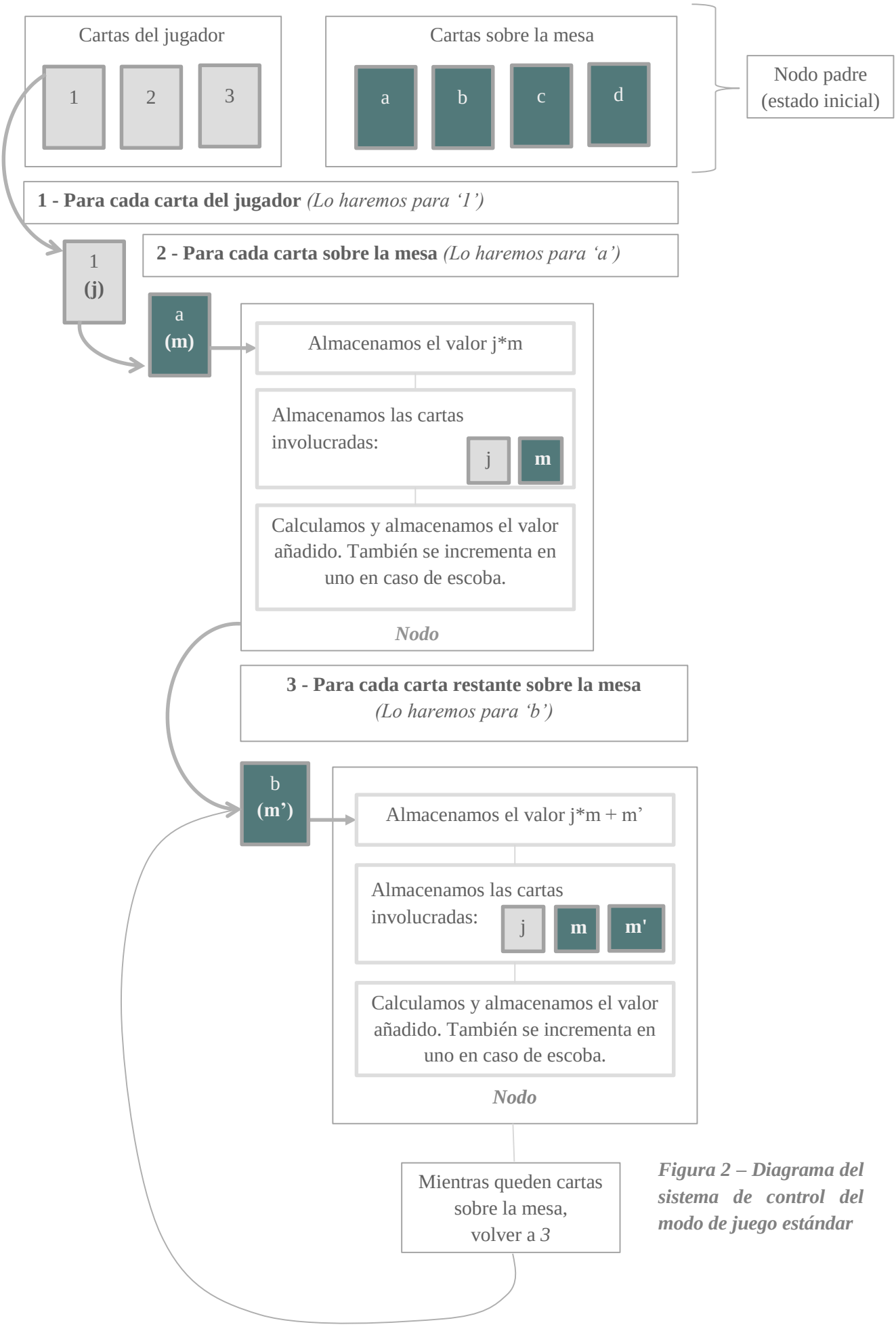


Figura 2 – Diagrama del sistema de control del modo de juego estándar

<u>Modo asociación</u>:

En el modo asociación, el objetivo es encontrar una combinación de cartas tal que multiplicando una de las del jugador por una o un grupo de cartas de las de la mesa y, opcionalmente, sumando cuantas cartas quiera de las restantes sobre la mesa, se obtenga un múltiplo de 5 no divisible entre 2 (5, 15, 25,…).

Algoritmo:

1. Para cada carta del jugador (j)
    a. Para cada carta de la mesa (m)
        i. Se multiplica j por m
        ii. Se almacena el valor resultante (j*m)
        iii. Se almacenan las cartas involucradas (2 en esta etapa) en una lista.
        iv. Se calcula el valor añadido de la jugada (tiene en cuenta si hay o no 5 o 7 de oros, etc.), sumando el valor añadido de las cartas involucradas.
        v. Si *m* es la última carta que queda sobre la mesa, se añade un punto al valor añadido acumulado de la jugada (pues se hace escoba).
        vi. A continuación, se realiza A y B:

            ***Nota**: los dos siguientes pasos se hacen en paralelo, uno no depende del otro. El objetivo es expandir el árbol por un lado como suma de cartas y por el otro como producto (y luego sumando).*

A: Para cada carta restante de la mesa (m'):

1. Se suma el valor m' al resultado anterior: j*m + m'
2. Se almacena el valor resultante
3. Se calcula el valor añadido de la jugada (incluyendo el de escoba).
4. Para cada carta restante sobre la mesa (m''), se vuelve a realizar A, iterativamente, hasta que no haya cartas sobre la mesa.
5. Se realiza B

B: Para cada carta restante sobre la mesa (m'):

1. Se suma el valor j*m', quedando: j*m + j*m' = j(m+m')
2. Se almacena el valor resultante
3. Se calcula el valor añadido de la jugada (incluyendo el de escoba).
4. Para cada carta restante sobre la mesa (m'''), se realiza A.

Ejemplo gráfico:

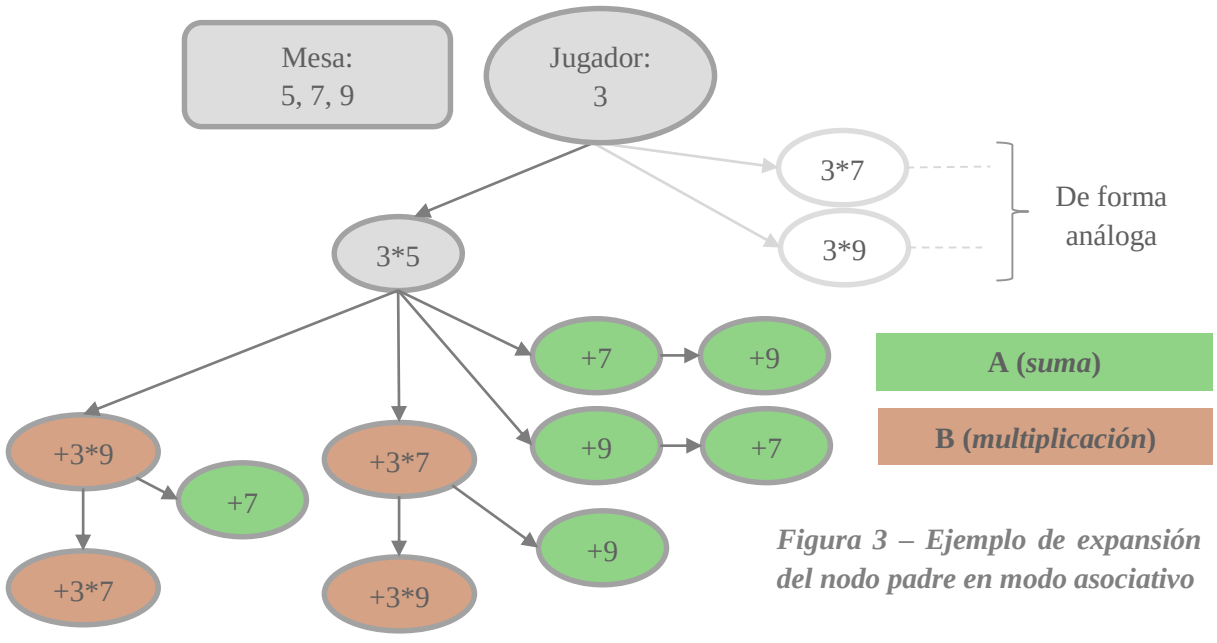


Figura 3 – Ejemplo de expansión del nodo padre en modo asociativo

Llegados a este punto, tenemos el árbol expandido en todas las posibles combinaciones: por tanto, esto incluye jugadas válidas (el valor resultante es 5, 15, 25,…) y jugadas no válidas (el valor resultante no es un múltiplo de 5 no divisible entre 2).

En el ejemplo gráfico superior, únicamente hemos expandido la carta "3" del jugador con la "5" de la mesa. Faltaría expandir "3" con "7" y "3" con "9", de forma totalmente análoga a la expansión de "3" con "5".

Con el objetivo de no complicar innecesariamente el árbol, si únicamente tenemos en cuenta la parte que hemos expandido en el ejemplo gráfico y asumimos que todas las cartas son oros, tendríamos:

Combinación	Válido (5, 15, 25,…)	Valor añadido
3*5 = 15	**SÍ**	0.5+1.5 = **2**
3*5 + 7 = 22	NO	---
3*5 + 7 + 9 = 31	NO	---
3*5 + 9 = 24	NO	---
3*5 + 9 + 7 = 31	NO	---
3*(5+7) = 36	NO	---
3*(5+7) + 9 = 45	**SÍ**	0.5+1.5+2.0+0.5+1.0 = **5.5**
3*(5+7+9) = 63	NO	---
3*(5+9) = 42	NO	---
3*(5+9) + 8 = 50	NO	---
3*(5+9+7) = 63	NO	---

Tabla 2 – Ejemplo de expansión en modo asociativo

Por lo tanto, de esta rama de expansión nos quedaríamos con la combinación 3*(5+7) + 9, cuyo valor añadido es 5.5 (nótese que se suma un punto más al valor añadido porque utiliza todas las cartas que hay sobre la mesa, es decir, la jugada hace escoba). Faltaría realizar lo mismo para las 2 expansiones que faltan y quedarnos con la combinación que mayor valor añadido ofrezca. En el siguiente apartado se detalla el algoritmo para esta elección de la solución.

2.4.2.3 Elección de la solución

La elección de la solución consiste en recorrer el árbol e ir almacenando la jugada que nos aporte el máximo valor añadido siendo una jugada válida. Es válido tanto para el modo de juego estándar como para el de asociación.

Algoritmo:

1. Reservar memoria para almacenar la jugada más ventajosa e inicializar su valor añadido a cero (cualquier jugada válida será más ventajosa - *tendrá mayor valor añadido* - que la inicial).
2. Para el nodo objeto de análisis
    a. Comprobar si es una jugada válida (valor resultante igual a 5, 15, 25,…)
        i. En caso de serlo → Compara si el valor añadido es mayor que el de la jugada almacenada
            1. Sí lo es → Lo remplaza
            2. Si no lo es → No hace nada
            3. Se ejecuta el algoritmo para cada uno de sus nodos hijo (si existen)

ii. En caso de no serlo:
    1. Se ejecuta el algoritmo para cada uno de sus nodos hijo

Finalmente, la solución será la jugada almacenada (que es la que mayor valor añadido tiene). Si su valor añadido es cero significará que no hay ninguna jugada válida y que, por tanto, el jugador ha de poner una de sus cartas sobre la mesa. La selección de la carta que se pondrá sobre la mesa se estudia a continuación.

Niveles de dificultad:

El nivel de dificultad del juego, recordemos que podía ser fácil, medio o difícil, se tiene en cuenta en cuenta en este punto: elección de la solución.

Hay dos maneras de implementar este comportamiento:

1. A medida de que se va realizando la búsqueda por cada nodo, asignamos distintas probabilidades de selección de la mejor jugada entre las dos posibles (la almacenada previamente y la nueva que acabamos de descubrir), por ejemplo:
    - ***Difícil***: nos quedamos el 100% de las veces con la mejor solución.
    - ***Medio***: nos quedamos el 50% de las veces con la mejor solución.
    - ***Fácil***: nos quedamos el 0% - 10% de las veces con la mejor solución.

    La ventaja de este método es que no implica utilizar memoria adicional.

2. Guardar todas las soluciones en una lista, ordenarlas por su valor añadido y luego elegir una de ellas en función del nivel de dificultad. Implica utilizar algo más de memoria, aunque despreciable debido a que el posible número de soluciones por jugada rara vez superará la decena.

En el caso de 2 jugadores siempre utilizaremos la segunda opción, guardar en una lista todas las soluciones, ya que necesitamos esta información para maximizar la puntuación hasta el siguiente reparte, como veremos en el apartado 2.4.3.2 – Elección de la mejor baza.

2.4.2.4 Elección de la carta que se desecha

En caso de no poder realizar ninguna jugada, el jugador deberá poner sobre la mesa una de sus cartas. Para ello y en ese caso (sin conteo), se desechará la carta de menor valor añadido (ver sección 2.4.1 - Carta).

En caso de empate se seleccionará una de ellas al azar, excepto en el siguiente caso:

- Si el valor añadido de las cartas es 0.5 y una – *o varias* - de ellas es un oro y otra – *o varias* - un siete de un palo distinto a oros (no hay más combinaciones de cartas distintas que ofrezcan el mismo valor añadido), se desechará el siete si disponemos de más del 50% de sietes y el oro en caso contrario.

2.4.3 Algoritmo 2 jugadores: con conteo

Una vez analizado el algoritmo sin conteo para el caso de 3 o 4 jugadores, estamos en condiciones de estudiar el caso más avanzado, el de 2 jugadores, en el que utilizaremos conteo para sacar toda la ventaja posible con la información disponible.

Únicamente se van a analizar las partes que difieren del algoritmo sin conteo, todo lo demás se supondrá análogo al caso anterior.

Utilizaremos el conteo para maximizar el beneficio en dos situaciones:

1. Cuando haya varias alternativas para hacer baza (varias soluciones posibles), en lugar de maximizar la puntuación de esa baza trataremos de maximizar la puntuación total de las siguientes bazas hasta el siguiente reparte. Excepto en un caso:
    - El contrario acaba de dejar una carta sobre la mesa. En este caso se buscará hacer baza con la carta que ha dejado el contrincante (y otras cartas que haya sobre la mesa), ya que eso significa que en el siguiente turno nuestro contrincante tampoco va a poder realizar ningún movimiento y se va a ver obligado a dejar otra carta sobre la mesa.
2. Cuando no sea posible hacer baza (sumar 5, 15, 25,…) y haya que depositar una carta sobre la mesa.

2.4.3.1 Conteo de cartas

En primer lugar, estableceremos las estructuras de datos utilizadas para realizar el conteo de cartas. Buscamos conocer qué cartas no tiene nuestro rival, para ello utilizaremos dos listas de cartas adicionales:

- ***Adversario***: lista de cartas en poder del adversario, pero no con las que pueda jugar, sino las que ya son suyas. Se utilizará para mejorar la elección de la carta depositada sobre la mesa (por ejemplo, si tiene cinco oros y un siete y nosotros tenemos un siete y un oro – *de un palo distinto a oros* - , depositaremos el oro para evitar que tenga más del 50% de sietes).
- ***Posibles***: lista de posibles cartas que puede tener el adversario para jugar en un determinado momento. Al comienzo de la partida esta lista será mayor e irá haciéndose más pequeña a medida de que se vayan descubriendo cartas. Para construir esta lista se empezará con toda la baraja y se irán eliminando las cartas que son descubiertas a lo largo de la partida.

Además de estas listas, almacenaremos si el contrincante ha depositado una carta sobre la mesa en la última jugada, ya que esta información nos será útil a la hora de elegir qué jugada realizar.

2.4.3.2 Elección de la mejor baza

Como hemos introducido anteriormente, utilizaremos el conteo para elegir la mejor solución en caso de que haya múltiples soluciones, pero no buscaremos maximizar la puntuación de la actual baza sino maximizar la puntuación hasta el siguiente reparte.

Para ello, simularemos para el peor de los casos (el contrincante siempre tiene las mejores cartas posibles, es decir, las de mayor valor añadido que todavía no han sido descubiertas) todas las posibles jugadas hasta que no le queden cartas a ninguno de los dos jugadores, y seleccionaremos aquella jugada que maximiza el valor añadido obtenido.

A continuación se detalla el algoritmo:

1. Se simula el estado inicial (búsqueda de posible soluciones tradicional) → **Obtenemos una serie de soluciones**. Calculamos el $VA_{Jugador}$ (valor añadido) para cada solución y lo almacenamos.
2. Si el contrincante ha depositado una carta sobre la mesa en la jugada anterior:
    - Buscamos las posibles soluciones que incluyan la carta que acaba de dejar el contrincante (que será la última en la lista de cartas sobre la mesa) y si hay alguna solución que cumpla con esto, paramos el algoritmo y nos quedamos con esa solución. En caso de múltiples soluciones que impliquen a la carta que ha dejado el contrincante, elegiremos la jugada de mayor valor añadido. En caso contrario (ninguna solución implica a la carta que ha dejado el contrincante), saltamos al paso 3.

    Si no, realizamos el paso 3.

3. **Para cada solución del paso anterior**, nos ponemos en el peor de los casos: el contrincante, quien tiene N cartas en su poder (sabemos cuántas cartas pero no qué cartas), tiene las de mayor valor añadido de la lista de posibles cartas. Para cada posible solución del paso 1, **se simulan las posibles jugadas del contrincante y**, para cada una de ellas se calcula el $VA_{Contrincante}$. Además, **calculamos el valor añadido neto que conseguimos** para cada caso: $VA_{Neto} = VA_{Jugador} - VA_{Contrincante}$ (El valor añadido del jugador viene del paso 2).
4. **Mientras le queden cartas a los jugadores, se vuelven a realizar los pasos 3 y 4** para cada posible jugada, expandiendo así un árbol de jugadas, donde para cada jugada se ha de expandir un árbol de soluciones, tal y como hemos hecho en el caso de 3 y 4 jugadores.
5. Una vez que el árbol de jugadas está expandido, **nos quedamos con la rama que mayor valor añadido neto nos aporta y realizamos la jugada correspondiente a dicha rama** (es decir, el primer movimiento de dicha rama).

Nota: dada la restricción de ciclos de reloj impuesta por las arquitecturas a las que este juego va destinado (procesadores móviles de baja y media potencia), se puede parar la simulación de jugadas transcurridos X milisegundos (500ms, por ejemplo) independientemente de si ha terminado o no la simulación de todas las posibles jugadas. Así cumpliríamos con los requisitos de tiempo real mientras que al mismo tiempo llegaríamos a una buena solución ya que, en el peor de los casos, siempre maximizaríamos nuestro valor añadido durante una serie de jugadas.

2.4.3.3 Elección de la carta que se desecha

El segundo escenario en el que podemos sacamos partido del conteo de cartas es a la hora de elegir la carta que vamos a desechar, en caso de tener varias posibilidades. Para maximizar el beneficio se elegirá la carta que menos nos va a perjudicar y que menos va a beneficiar a nuestro contrincante, para ello utilizaremos la información de la lista de posibles cartas.

En primer lugar, ordenaremos las cartas que tenemos para jugar de menor a mayor valor añadido. Si no utilizásemos conteo de cartas, desecharíamos la de menor valor añadido (salvo ante empate entre sietes y oros, como ya hemos indicado anteriormente en la sección 2.4.2.4),

pero en esta ocasión utilizaremos la información del conteo para también tratar de minimizar la ventaja que representa la carta elegida para el contrincante.

Esto se hará únicamente si hay más de una carta con el mínimo valor añadido, ya que daremos mayor prioridad a nuestro beneficio que a la pérdida de beneficio del contrincante.

En caso de empate de valor añadido, se seleccionarán todas estas cartas y se aplicarán la siguiente reglan, en orden:

- Si el valor añadido de las cartas es 0.5 y una – *o varias* - de ellas es un oro y otra – *o varias* - un siete de un palo distinto a oros (no hay más combinaciones de cartas distintas que ofrezcan el mismo valor añadido), se verificará si el contrincante tiene más de la mitad de los oros en su poder. En caso de que no, se pondrá sobre la mesa el siete y nos quedaremos con el oro. Si tiene más del 50% de oros, nos da igual qué carta desechar y se seleccionará al azar.
- Si el valor añadido mínimo de las cartas es distinto de 0.5, se desechará una de ellas al azar.

3 Tecnologías para la implementación

Una vez analizado el aspecto algorítmico del proyecto, estamos en condiciones de seleccionar la tecnología que se utilizará para su implementación, para lo cual buscaremos y evaluaremos las distintas alternativas disponibles.

Como resumen de los requisitos técnicos del proyecto (para más información ver el apartado 1.2 – Requisitos), las principales restricciones son:

- Multiplataforma: compatible con *Android* [2] y *Microsoft Windows* [1].
- Soporte para terminales móviles poco potentes.
- Tamaño pequeño del paquete resultante menor que $10\ MB$.

Como introducción y dado que el programa resultante ha de ser compatible con *Android* y *Microsoft Windows*, tenemos disponibles las siguientes opciones:

1. Utilizar un conjunto de tecnologías escritas en *Java* [9], que ofrezcan soporte tanto para Windows como para Android.
2. Utilizar código nativo (librerías en C/C++ que soporten tanto Windows como Android)
3. Realizar dos implementaciones distintas, una para plataforma. Desaconsejable.
4. Utilizar motores de alto nivel. Demasiado pesado para terminales poco potentes, además de un tamaño de paquete considerablemente alto.

Las opciones preferidas serán la 1 y la 2, por lo que únicamente se recurrirá a la 3 o 4 en el improbable caso de que no exista ninguna tecnología multiplataforma y de bajos requerimientos computacionales que se adapte a nuestro proyecto.

3.1 Posibles soluciones

Tras realizar una búsqueda exhaustiva de posibles tecnologías de implementación, se han seleccionado las siguientes tecnologías candidatas:

- ***Cocos2D-x*** [10]: Framework multiplataforma para desarrollo de juegos e interfaces 2D. Muy usado en la industria de los juegos móviles, permite utilizar código nativo compilando el mismo código para distintas plataformas. Capaz de ser ejecutado en terminales de bajas prestaciones gracias al uso de *OpenGL ES 1.1/2.0* [5] y código nativo (considerablemente más rápido que Java en el caso de Android, especialmente para la expansión de árboles de nuestro algoritmo, caso que nos ocupa). Software libre.

- ***Ogre3D*** [11]: Motor 3D con una API orientada a objetos. También usado extensamente en la industria de los videojuegos, permite utilizar el mismo código para crear juegos y aplicaciones multiplataforma. En el caso de Android, utiliza la versión 2.0 de *OpenGL ES*, por lo que requiere terminales algo más nuevos que en el caso de Cocos2D-x, además es un motor 3D y esto conlleva una gran carga computacional extra (la forma en que se organizan las escenas, materiales, cambios de estados de *OpenGL*, etc…). Software libre.

- ***Bonzai engine*** [12]: Motor 3D escrito en Java y multiplataforma. No es muy conocido, aunque tiene buenos ejemplos. Es un motor 3D, por lo que es mucho menos ligero que un motor 2D. Comercial.

- ***Unity3D*** [13]: Motor 3D comercial, lógica programable a través de scripts (utiliza la máquina virtual *Mono*, por lo que la lógica puede ser escrita en C#, *Javascript*,…). Necesita *OpenGL ES 2.0* [5] y es comercial. Además, el tamaño del ejecutable resultante es considerablemente grande.

3.2 Selección de la solución

A continuación se muestra una tabla con los pros y los contras de cada tecnología:

Tecnología	2D nativo	Comercial	Multiplataforma	Tamaño del ejecutable	OpenGL ES
Cocos2D-x	Sí	No	Sí	Bajo	1.1/2.0
Ogre3D	No	No	Sí	Alto (>15Mb)	2.0
Bonzai	No	Sí	Sí	Medio	2.0
Unity3D	No	Sí	Sí	Alto	2.0

Tabla 3 – Tabla comparativa de tecnologías de implementación

Como podemos observar, la opción más adecuada para nuestro proyecto es ***Cocos2D-x*** [10]. Además de ser el único motor específicamente diseñado para gráficos 2D, también es el único que soporta hardware más antiguo y además el tamaño del paquete resultante es el más pequeño.

El resto de tecnologías están orientadas principalmente a gráficos 3D, aunque es perfectamente posible utilizar estos motores para gráficos 2D. No obstante y para este proyecto, estas tecnologías quedan descartadas.

3.3 Estudio detallado de la solución seleccionada

La tecnología de implementación seleccionada es el motor gráfico 2D *Cocos2D-x*, que como ya hemos introducido anteriormente es un motor ligero, flexible, libre, multiplataforma y disponible en varios lenguajes de programación.

Hay disponible una gran cantidad de documentación y existe una enorme comunidad de usuarios, pues se estima que el 30% de los juegos de plataformas móviles están desarrollados con este motor, por lo que es una tecnología madura y probada.

En nuestro caso, utilizaremos C++ para desarrollar el juego. El código será el mismo para ambas plataformas (*Microsoft Windows* [1] y *Android* [2]), exceptuando algunas pequeñas partes relacionadas con la gestión de eventos de entrada (pantalla táctil vs ratón, etc.).

La versión para Windows se compilará con *Microsoft Visual Studio 2012 Express* [14], la versión gratuita del IDE por excelencia para plataformas Windows, como se haría con cualquier otro programa escrito en C++.

Para la versión de Android se utilizará código nativo, ya que nuestro programa estará escrito en C++ y no en *Java* [9], para lo cual utilizaremos el entorno de desarrollo *Eclipse* [15] y el *NDK* (*Native Development Kit* [16]) de Android.

Finalmente y aunque en un principio quede fuera de los márgenes del proyecto, *Cocos2D-x* soporta muchas más plataformas, entre las que podemos destacar: *iOS* [17], *Blackberry OS* [18], *Windows Phone* [19], *Tizen* [20],… Por lo que siempre queda la posibilidad de crear versiones para estos sistemas operativos, dado que básicamente el esfuerzo requerido sería preparar el entorno de desarrollo del sistema operativo en cuestión y compilar nuestro proyecto.

4 Conclusiones de la etapa de investigación

Una vez hemos realizado el estudio de la Inteligencia Artificial requerida para nuestro proyecto y seleccionado la tecnología de implementación, estamos en condiciones de extraer una serie de conclusiones a modo de resumen.

Dado que esta etapa del proyecto se ha dividido en dos partes claramente diferenciadas: I.A. y tecnología de implementación, mantendremos esa división lógica en las conclusiones.

4.1 Inteligencia Artificial

- Utilizamos la teoría de resolución de problemas para plantear los algoritmos necesarios, basados en la expansión de nodos (estados del juego) creando árboles.
- Hay dos modos distintos de juego, estándar y asociación, además de tres niveles de dificultad distintos: fácil, medio, difícil.
- Además del punto anterior, diferenciamos dos modos de inteligencia distinta: con conteo (2 jugadores) y sin conteo (3 o 4 jugadores).
- En total, se han planteado dos algoritmos distintos, correspondientes a los casos con conteo y sin conteo. El resto de comportamientos se implementan en las reglas y sistema de control.
- El algoritmo de juego sin conteo trata de maximizar la puntuación de la baza actual en función de las cartas que posee el propio jugador. Se realiza una expansión de estados para todas las posibles jugadas que puede hacer el jugador y se selecciona la más ventajosa, pero no se simulan estados posteriores del juego.
- El algoritmo con conteo no trata de maximizar la puntuación de la baza actual, sino que trata de maximizar la puntuación hasta el siguiente reparte. Para ello, además de simular cada baza, se simulan las siguientes bazas hasta que ambos jugadores se quedan sin cartas. Una vez hecho esto, se realiza la jugada más ventajosa.
- En el caso con conteo, siempre nos ponemos en la peor situación, es decir, asumimos que el contrario siempre tiene las mejores cartas posibles.
- Se utiliza el concepto de valor añadido para cuantificar cuán buena es una carta.
- En caso de no poder realizar un movimiento, la carta elegida para poner sobre la mesa es aquella que nos proporciona una menor pérdida (menor valor añadido) y además, en el caso con conteo, también se tienen en cuenta las cartas que puede tener el contrario para realizar la elección en caso de empate de valor añadido.

4.2 Tecnología de implementación

- Para cumplir con los requisitos del proyecto, la tecnología seleccionada tenía que ser compatible con Windows y Android además de ser ligera y orientada a gráficos 2D.
- De entre las cuatro opciones que hemos evaluado: *Cocos2D-x* [10], *Ogre3D* [11], *Bonzai Engine* [12] y *Unity3D* [13], nos hemos decantado por la primera de ellas: ***Cocos2D-x***.
- Cocos2D-x es multiplataforma, ligera, soporta hardware relativamente antiguo y además es software libre.
- Aproximadamente el 30% de juegos móviles están desarrollados con esta tecnología.
- La implementación se realizará en C++.
- En Android, se utilizará el NDK (*Native Development Kit* [16]), ya que nuestro proyecto no estará hecho en *Java* [9] sino en C++.

5 Diseño del juego y sus distintos estados

Una vez concluida la etapa de diseño teórico de la aplicación estamos en condiciones de diseñar los distintos estados del juego.

De ahora en adelante, usaremos el término *estado (o pantalla)* de juego para hacer referencia a los distintos menús y vistas a los que podrá acceder el usuario dentro del juego.

En primer lugar, definiremos las distintas acciones que el usuario ha de ser capaz de realizar utilizando la interfaz de usuario del juego y una vez definidas diseñaremos dicha interfaz de usuario.

A través de la interfaz de usuario, el usuario debe ser capaz de:

- Seleccionar el jugador con el que jugar las partidas. De esta manera, más de un usuario podrá jugar a la ESCOBA² en el mismo terminal y llevar una progresión distinta dentro del juego.
- Consultar el progreso de cada jugador, visualizando el número total de juegos ganados, perdidos y una puntuación total que asigna distintos pesos según tipo de juego ganado (simple – 1 punto - o torneo – 2 puntos -). Esta puntuación será la utilizada para desbloquear los distintos retos.
- Iniciar una partida simple, seleccionando el número de jugadores (de 2 a 4), el modo de juego (simple o asociativo) y el nivel de dificultad (fácil, medio o difícil).
- Iniciar un torneo donde se pueda seleccionar el número de rondas (de 2 a 20), el número de jugadores (de 2 a 4), el modo de juego (simple o asociativo) y el nivel de dificultad (fácil, medio o difícil). El usuario que gana el torneo es el usuario que más puntos acumula a través de las distintas rondas del mismo.
- Consultar el siguiente reto disponible para desbloquear. Cada reto desbloqueará una nueva baraja de juego.
- Consultar las reglas del juego mediante un menú de ayuda.
- Salir del juego.

5.1 Pantallas del juego (estados)

Una vez que hemos definido las distintas acciones que el usuario ha de ser capaz de realizar, estamos en condiciones de diseñar los distintos estados necesarios para el juego ESCOBA².

A continuación se listan los distintos estados que se han decidido implementar para ofrecer la funcionalidad requerida:

- Menú principal.
- Menú de configuración de partida.
- Vista de juego.
- Menú de configuración de torneos.
- Estadísticas.
- Ayuda.

Seguidamente se estudia en detalle el proceso de diseño de cada estado.

5.1.1 Menú principal

El menú principal es, como su propio nombre indica, la primera pantalla visualizada por el usuario una vez que éste arranca el juego y desde el cual se accede al resto de estados del juego.

El menú principal incluye los siguientes botones:

- **Nueva partida**: inicia el menú de configuración de la partida (Ver sección 5.1.2 – Menú de configuración de la partida).
- **Nuevo torneo**: inicia el menú de configuración de torneos (Ver sección 5.1.4 – Menú de configuración de torneos).
- **Estadísticas**: inicia la vista de estadísticas (Ver sección 5.1.5 – Estadísticas).
- **Ayuda**: inicia la vista de ayuda (Ver sección 5.1.6 – Ayuda).
- **Salir**: cierra la aplicación.

Además de los distintos botones listados arriba, el menú principal incluye elementos de interfaz para seleccionar el jugador y visualizar el reto actual, detallados a continuación.

5.1.1.1 Selección del jugador

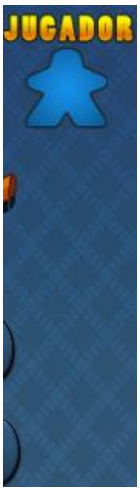
El sistema de jugadores se ha implementado de una manera muy visual y directa: hay cinco jugadores distintos disponibles, cada uno con su propio progreso (estadísticas). Cada jugador está identificado con un color: rojo, azul, verde, amarillo y naranja.

El jugador actual se muestra en la esquina superior derecha. Para cambiar de jugador simplemente hay que pulsar sobre esta área y se desplegarán el resto de jugadores, pulsando sobre cualquiera de ellos se seleccionará dicho jugador.

A continuación se muestra la secuencia de selección de jugadores:

1. El jugador rojo está seleccionado

2. El usuario pulsa sobre el jugador rojo



3. El usuario selecciona el jugador azul

Figura 4 – Secuencia de selección de jugadores

5.1.1.2 Retos

El sistema de retos muestra de forma automática un icono parpadeante en la esquina inferior derecha de la pantalla en función de la progresión del jugador seleccionado. Si no hay ningún reto disponible – es decir, ya se han superado todos los retos – no se mostrará dicho icono.

Al pulsar sobre este icono se mostrará el reto actual y qué baraja se desbloquea si el usuario llega a acumular el número de puntos requerido para superar el reto.

A continuación se muestra una captura de pantalla de la zona de retos:

1. Área de retos

2. Cuando el usuario pulsa sobre el área de retos se muestra el reto actual

Figura 5 – Detalle del área de retos

5.1.1.3 Versión final

A continuación se muestra una captura del menú principal de la ESCOBA²:

Figura 6 – Versión final del menú principal

5.1.2 Menú de configuración de la partida

El menú de configuración de la partida le permite al usuario seleccionar los siguientes parámetros de la partida:

- Número de jugadores: 2, 3 o 4.
- Modo de juego: estándar o asociativo.
- Nivel de dificultad: fácil, medio o difícil.

Es funcionamiento es el siguiente: en primer lugar, el usuario selecciona el número de jugadores: 2, 3 o 4; en segundo lugar se selecciona el modo de juego junto con la dificultad, a modo de botón, que iniciará la partida.

En todo momento está disponible un botón en la parte inferior para volver al menú principal.

A continuación se muestra una captura de pantalla del menú de configuración de la partida en el juego ESCOBA²:

Figura 7 – Menú de configuración de la partida

5.1.3 Vista de juego

Una vez introducidos el menú principal y el menú de configuración de la partida, estamos en condiciones de describir el corazón del juego: la vista de juego.

La vista de juego es la interfaz de usuario a través de la cual el usuario juega las partidas en la ESCOBA².

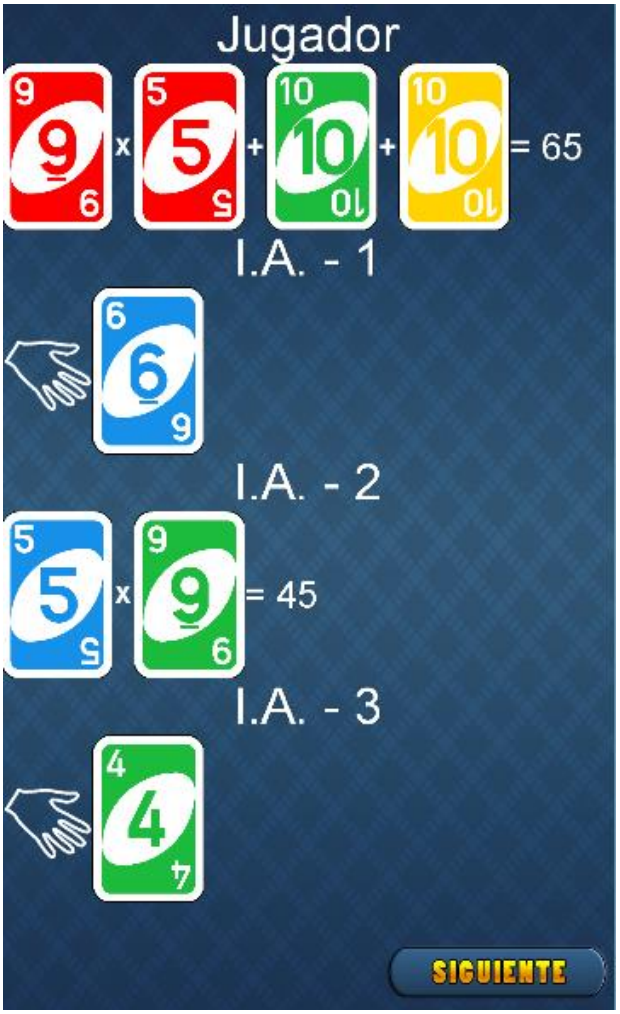
El funcionamiento de la vista de juego es el siguiente:

1. Al iniciar la partida, las cartas del usuario se muestran en la mitad inferior mientras que las cartas que hay sobre la mesa se muestran en la parte superior de la pantalla.
    a. El usuario selecciona una de sus cartas y a continuación selecciona una de las de la mesa. El resultado provisional de la jugada (multiplicación de las cartas) se mostrará en la pantalla únicamente en los niveles de dificultad fácil y medio.
    b. A continuación el usuario podrá seleccionar, si así lo desea, cuantas cartas sobre la mesa queden para sumar al resultado actual con el objetivo de obtener un resultado de 5, 15, 25, …
    c. En todo momento habrá tres botones disponibles:
        i. *Reset*: reinicia la jugada deshaciendo todos los cambios realizados.
        ii. *Jugar*: si la jugada es válida (resultado mostrado en verde), se realiza la jugada. Este botón únicamente se activa si la jugada es válida.
        iii. *Pasar*: si el usuario no puede realizar ningún movimiento válido, pulsa este botón para depositar una carta sobre la mesa.

En todo momento se muestra el reparte actual en la parte inferior de la pantalla. El número de repartes totales depende del número de jugadores: 6 repartes para 2 jugadores, 4 repartes para 3 jugadores y 3 repartes para 4 jugadores.

Figura 8 – Vista de juego

2. Cuando el usuario pulsa en *Jugar* o *Pasar*, realizará la jugada (o descarte) y a continuación se mostrará una animación con el resultado de las jugadas que han realizado los jugadores: el usuario y los contrincantes (inteligencias artificiales).



En este caso los jugadores 2 y 4 han depositado una carta sobre la mesa, el jugador número 1 (humano) ha realizado una juagada que suma 65 y el jugador número 3 una que suma 45.

Figura 9 – Jugadas realizadas en el turno actual

3. Cuando el usuario pulsa en *Siguiente*, volveremos a la vista número 1 hasta que se realizan todas las jugadas y ya no haya más cartas por repartir ni más cartas sobre la mesa.

   Una vez que esto ocurre (la partida finaliza), veremos el resultado final de la partida, con la puntuación que han obtenido todos los jugadores desglosada y un mensaje distinto en función de si el jugador gana, pierde o empata. Los puntos obtenidos por el ganador se mostrarán resaltados en verde. Además también se mostrará el nombre del jugador ganador. En caso de empate no se remarcará ningún jugador con el color verde y se mostrará el mensaje de "*Tablas*".

A continuación se muestra una captura de la pantalla de resultados dentro de la vista de juego para las 3 opciones posibles: el jugador humano gana, pierde o se produce un empate:

1. *El jugador gana*

2. *El jugador pierde*

3. *Se produce un empate*

Figura 10 – Pantallas de resultados

Cabe destacar que durante todo el transcurso de la partida, el usuario tendrá disponible un botón para rendirse y volver al menú principal. Dicho botón abrirá un diálogo de confirmación para evitar que la partida termine por error:

Figura 11 – Diálogo de confirmación de rendición

5.1.3.1 Modo de juego asociativo

El comportamiento de la vista de juego durante el modo asociativo es muy similar al del modo estándar, únicamente cambia el modo en el que se seleccionan las cartas para realizar la jugada.

Durante el modo asociativo, se muestra un botón con el símbolo '+'. Hasta que el usuario no pulsa dicho botón, las cartas que selecciona de las cartas sobre la mesa se incluyen en el factor de multiplicación. Una vez que el usuario pulsa sobre el botón, el resto de cartas que se seleccionan se suman al resultado final.

A continuación se muestra una captura de la vista de juego en modo asociativo:

El usuario a pulsado sobre su carta '3' y acto seguido a pulsado sobre las cartas '5' y '10'. Por tanto el resultado que obtiene es 3(5+10) = 45. Si ahora pulsa el botón '+' y pulsa sobre la carta '10' obtendría el siguiente resultado: 3*(5+10)+10=55.*

Figura 12 – Vista de juego en modo asociativo

5.1.4 Menú de configuración de torneos

Como ya se ha mencionado anteriormente, un torneo está formado por un conjunto de rondas. Al finalizar un torneo, se suman - para cada jugador - todos los puntos que han obtenido en las rondas jugadas y el jugador que gana el torneo es el jugador que mayor número de puntos ha acumulado durante las distintas rondas disputadas.

Por tanto, el menú de configuración de torneos permite al usuario seleccionar el número de rondas de las que constará el torneo. Una vez que el usuario ha elegido el número de rondas, tendrá disponible un botón para empezar a jugar.

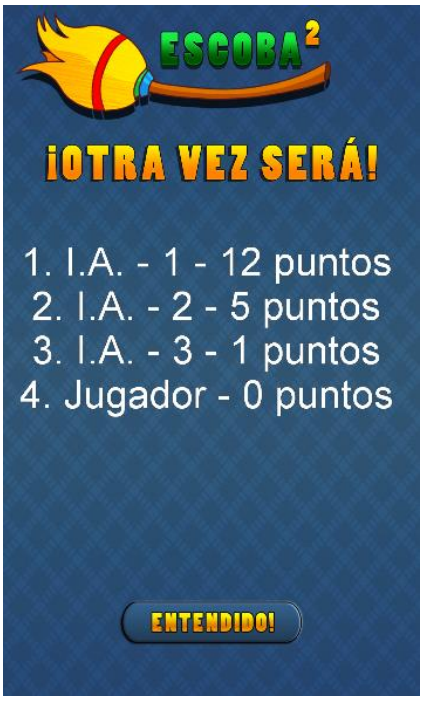
Además, en todo momento habrá un botón en la parte inferior de la pantalla para volver al menú principal.

La vista de juego durante los torneos es idéntica a la vista de juego durante un juego simple a excepción de mostrar en la esquina inferior derecha la ronda actual. Además, cuando se terminan todas las rondas del torneo se muestra el resultado final con los puntos acumulados por todos los jugadores y la clasificación de los jugadores.

A continuación se muestra una captura de pantalla tanto del menú de configuración de torneos como de la vista de juego durante un torneo y de la vista de resultado final del torneo:

1. Menú de configuración de torneos

2. Vista de juego durante un torneo. Nótese la información sobre la ronda actual en la esquina inferior derecha.



3. Vista de resultado final del torneo

Figura 13 – Pantallas del juego en modo torneo

5.1.5 Estadísticas

Una vez diseñados los distintos estados necesarios para jugar a todos los modos de juego de la ESCOBA², diseñaremos la vista de las estadísticas.

Las estadísticas han de ser rápidas de leer, pues han de indicar la progresión de cada jugador de forma sencilla. Por tanto, para cada uno de los cinco jugadores disponibles, se mostrará la siguiente información:

- Número de partidas ganadas (juegos simples o torneos).
- Número de partidas perdidas (juegos simples o torneos).
- Puntuación: 1 punto por cada juego simple ganado y 2 puntos por cada torneo ganado.

Además, se incluye un botón para volver al menú principal una vez que el usuario haya terminado de ver las estadísticas.

A continuación se muestra una captura de pantalla de la vista de estadísticas:

En la primera columna se muestran las partidas (simples o torneos) que ha ganado cada jugador. En la segunda columna se muestran las partidas (simples o torneos) que ha perdido cada jugador. Finalmente, en la tercera columna, se muestra la puntuación total: se obtiene un punto por cada juego simple ganado y dos puntos por cada torneo ganado. Esta puntuación total es la que se utiliza para desbloquear los distintos retos.

Figura 14 – Pantalla de estadísticas

5.1.6 Ayuda

Finalmente, queda por incluir en el juego una pequeña ayuda donde se expliquen tanto las reglas como cómo jugar al juego de la ESCOBA².

Para implementar esta vista de ayuda se ha optado por unas instrucciones que se van desplazando de abajo arriba y que el usuario puede situar donde quiera presionando y deslizando su dedo sobre la pantalla.

En esta vista de ayuda se explican las bases del jugo y cómo se ha de jugar una partida, además de incluir una sección de créditos. Así mismo y de igual forma en la vista de estadísticas, habrá un botón en la parte inferior de la pantalla para volver al menú principal.

A continuación se incluyen dos capturas de pantalla de la vista de ayuda:

Figura 15 – Pantalla de ayuda

5.2 Diagrama de flujo entre estados

A continuación se incluye un diagrama de flujo entre las distintas pantallas del juego:

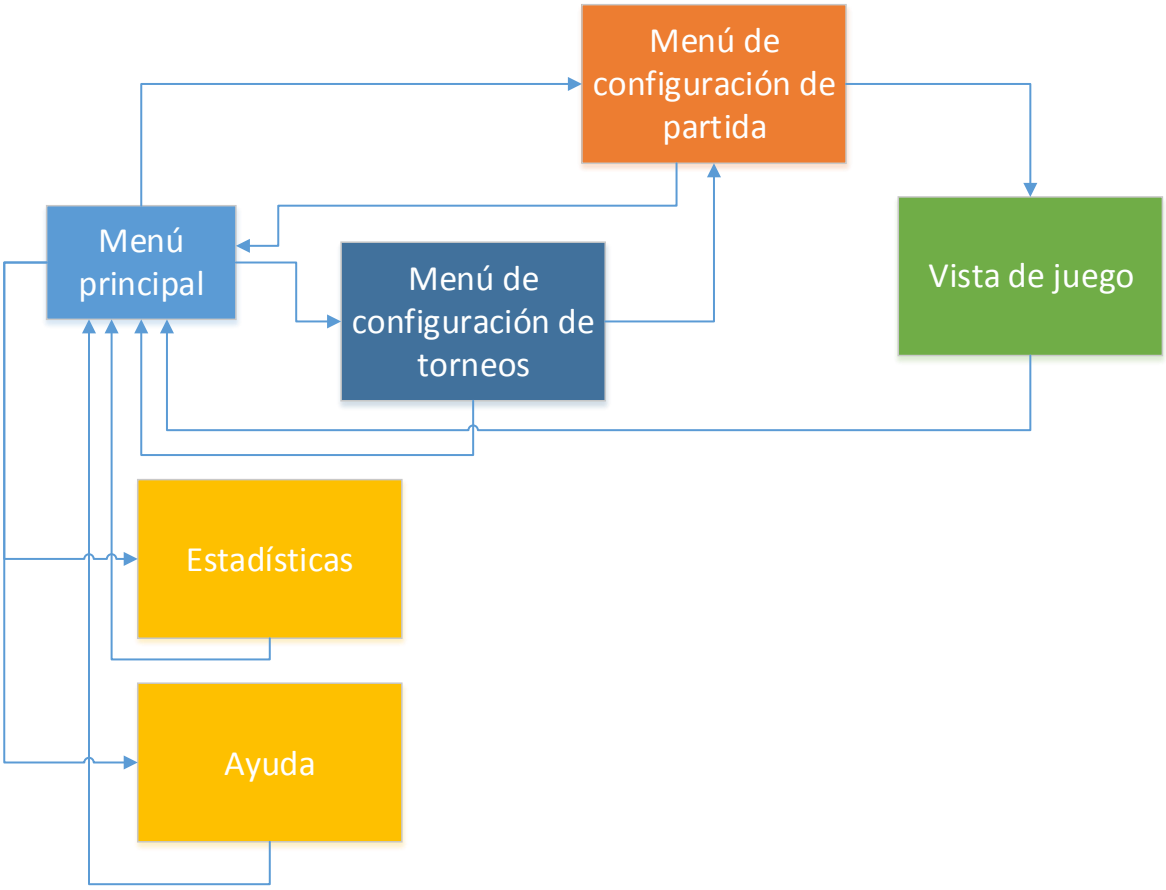


Figura 16 – Diagrama de flujo entre estados

6 Entorno de desarrollo

Una vez que hemos diseñado los algoritmos necesarios para el correcto funcionamiento del juego, decidido qué estados debe tener el juego, cómo han de estar organizados y seleccionado la tecnología de implementación que vamos a utilizar, estamos en posición de comenzar con el desarrollo software del proyecto, es decir, de comenzar con la implementación práctica.

En primer lugar, nos centraremos en la configuración y descripción del entorno de desarrollo que se va a utilizar para crear el ejecutable, tanto para la plataforma *Microsoft Windows* [1] como para la plataforma *Android* [2].

Se ha optado por una organización de proyecto compartida y multiplataforma, es decir, que se utilizará el mismo árbol de directorios y los mismos ficheros para el desarrollo de las distintas versiones del juego (Windows y Android). Para ello, se incluirán los distintos archivos de proyecto en el directorio padre, de forma que se pueda generar la aplicación final de forma sencilla y sin necesidad de modificaciones.

Cabe destacar que para generar la versión Android del juego, se utilizará el sistema operativo Microsoft Windows, por lo que todo el entorno de desarrollo será en esta misma plataforma, aunque estemos compilando el proyecto para Android.

La jerarquía de directorios que utilizaremos es la siguiente: un directorio padre que albergará todas las dependencias y el proyecto ESCOBA², que será el juego en sí mismo. Las dependencias se generarán en sus propios directorios y, salvo las cabeceras de las mismas, tanto las bibliotecas de vinculación (archivos **.lib*) como las bibliotecas dinámicas (archivos **.dll* en Microsoft Windows) serán copiadas a sus respectivas carpetas dentro del proyecto ESCOBA².

El árbol de directorios de nuestro proyecto, ESCOBA², es el siguiente:

- ***bin/*** Directorio donde se generan los ejecutables
  - ***release/*** Modo release
  - ***debug/*** Modo de depuración
- ***lib/*** Directorio donde se almacenan las bibliotecas de vinculación
- ***obj/*** Directorio intermedio, donde se genera el código intermedio antes de ser enlazado.
  - ***release/*** Modo release
  - ***debug/*** Modo de depuración
- ***include/*** Carpeta donde se almacenan las cabeceras del proyecto
- ***src/*** Directorio donde se almacena el código fuente del proyecto

Nota: todos los recursos necesarios para el juego (imágenes, sonidos,…) serán situados en carpetas pertenecientes a bin/release/ o bien /bin/debug/ y nunca en sitios externos.

6.1 Windows

Como ya hemos introducido anteriormente, las tecnologías que van a ser utilizadas para el desarrollo del proyecto en Microsoft Windows son:

- ***Cocos2D-x*** [10]
- ***Microsoft Visual Studio 2012 Express*** [14]

Una vez instalado *Visual Studio 2012*, nuestro IDE (entorno de desarrollo integrado: editor, compilador y depurador), estamos en condiciones de descargar y configurar Cocos2D-x, el motor gráfico que usaremos. La versión que utilizaremos es la 2.2.1 [21].

Para configurar y compilar Cocos2D-x, únicamente hemos de descomprimir el paquete en el mismo directorio que contiene el proyecto ESCOBA², quedando el siguiente árbol de directorios:

- **ESCOBA2**/bin/
  /obj/
  /…
- **cocos2d-x-2.2.1**/cocos2dx/
  /samples/
  /…

Una vez descomprimido, únicamente hemos de ejecutar el archivo ***build-win32.bat*** y automáticamente se generarán los binarios correspondientes, utilizando Visual C++ 2012 como compilador.

Finalmente, copiaremos los archivos generados en ***cocos2d-x-2.2.1/Release.win32*** a los respectivos directorios en nuestro proyecto ESCOBA² (los *.dll* a */bin* y los *.lib* a */lib*).

Para nuestro proyecto, necesitamos copiar:

- *glew32.dll + lib*
- *libcocos2d.dll + lib*
- *libtiff.dll + lib*
- *pthreadVCE2.dll + lib*
- *zlib1.dll*

Una vez hecho esto, abrimos la solución de Visual Studio de nuestro proyecto (***ESCOBA2.sln***) y generamos el proyecto.

6.2 Android

A continuación trataremos en detalle los pasos a seguir para compilar una versión compatible con la plataforma *Android* [2], utilizando la plataforma *Microsoft Windows* [1] para generar esta versión.

El *port* a la plataforma Android ha sido realizado una vez que la versión para Windows estaba madura: toda la lógica/I.A. y la mayoría de vistas ya terminadas y totalmente funcionales.

Las partes que todavía no han sido cerradas en esta etapa del desarrollo son:

- Almacenamiento de estadísticas – debido a la necesidad de guardar archivos en disco duro, lo cual no es, a priori, multiplataforma y se requieren implementaciones especializadas.
- Recursos gráficos (imágenes) – muy dependientes de la resolución y relación de aspecto de la ventana disponible.

Hay que recordar que nosotros vamos a compartir la mayor parte del código entre ambas plataformas (parte C++) y luego algunas partes (tales como la creación de la ventana, gestión de eventos o gestión de ficheros) necesitarán de una implementación específica dependiendo del sistema operativo (En *Java* [9] para Android y en C++ para Windows). Por suerte, gran parte de esta funcionalidad ya viene abstraída y adecuadamente implementada por *Cocos2D-x* [10] y por tanto no requerirá de esfuerzos adicionales.

6.2.1 Software requerido

Para generar una versión compatible con sistemas Android utilizando la plataforma Microsoft Windows necesitamos el siguiente software:

1. JDK (*Java Development Kit*) [22]
2. Android SDK [23]
3. Cygwin [24]
4. Android NDK [16]

6.2.1.1 JDK (Java Development Kit)

Tras instalar JDK [22] debemos crear una variable de entorno llamada *JAVA_HOME* y que contenga la carpeta de instalación del JDK, en nuestro caso: "*C:Files\Java\jdk1.7.0_55"*.

6.2.1.2 Android SDK

El SDK (*Software Development Kit*) de Android incluye las APIs y herramientas necesarias para desarrollar, probar y depurar aplicaciones para Android.

Nosotros utilizaremos el "ADT Bundle" (*Android Developer Tools* [25]), que incluye:

- IDE Eclipse [15] + plugin ADT.
- Herramientas del SDK.
- Herramientas de la plataforma Android.
- La última plataforma Android.
- La última imagen de Android disponible para el emulador.

El *ADT Bundle* [25] se distribuye como un archivo ZIP, el cual contiene todo lo necesario para desarrollar en *Android* [2]. En nuestro caso lo descomprimiremos en "*C:/adt-bundle-windwos-x86_64-20150321*".

Además, deberemos de crear una variable de entorno llamada *ANDROID_SDK* y que contenga "*C:/adt-bundle-windwos-x86_64-20150321/sdk".*

6.2.1.3 Cygwin

Cygwin [24] es una colección de herramientas que ofrecen en *Microsoft Windows* [1] una funcionalidad similar a la que podemos encontrar en sistemas *Linux* [26].

Al instalar *Cygwin* elegiremos los siguientes paquetes: *autoconf, automake, binutils, gcc-core, gcc-g**, gcc4-core, gcc4-g**, gdb, pcre, pcre-devel, gawk* y *make*.

Tras finalizar la instalación añadiremos a la variable de entorno *PATH* el directorio de instalación de binarios de *Cygwin*, en nuestro caso: "*C:\cygwin64\bin*".

6.2.1.4 Android NDK

Finalmente instalaremos el *Native Development Kit* [16] de Android, para poder compilar aplicaciones escritas en C++ para la plataforma Android. Al igual que el *SDK* [23], se trata de un archivo ZIP que hemos de descomprimir.

Una vez descomprimido, crearemos una nueva variable de entorno llamada *NDK_ROOT* y que contenga el directorio en el que hemos descomprimido el *NDK*, en este caso "*C:\android-ndk-r9d-windows-x86_64*".

Nota importante: hemos de asegurarnos de que ninguno de los directorios – *tanto el que contiene el NDK como el que contiene el proyecto en sí* – contiene espacios. Esto es una limitación impuesta por el *toolchain* de Android, que utiliza GNU Make y éste no maneja correctamente los directorios que contienen espacios.

6.2.2 Creación del proyecto Eclipse

Finalmente, tras instalar todo el software necesario, estamos en condiciones de crear el proyecto Eclipse para compilar nuestro proyecto para la plataforma Android.

En primer lugar, crearemos un nuevo *workspace* de *Eclipse* [15] para nuestro proyecto. Una vez creado, tendremos en primer lugar que importar el proyecto localizado en '*cocos2dx/platform/android'*.

Para nuestro proyecto, utilizaremos como base uno de los proyectos de ejemplo que viene con *Cocos2D-x* [10]. Únicamente sustituiremos nuestro código fuente por el del ejemplo, así como los recursos (imágenes, etc.).

7 Arquitectura software

Tras describir el entorno de desarrollo que utilizaremos para desarrollar el juego ESCOBA², realizaremos un estudio exhaustivo del diseño de software que se ha utilizado.

En primer lugar y como ya ha sido mencionado anteriormente, el juego se desarrollará íntegramente en C++, tratando en todo momento de cuidar el diseño desde un punto de vista global así como buscar la optimización y el rendimiento de las pequeñas partes.

Para ello, se utilizará un patrón de diseño orientado a objetos para las clases que darán forma a la implementación algorítmica y la parte gráfica y, cuando sea necesario (mayormente en los algoritmos de resolución del problema), un patrón de diseño orientado a datos buscando el máximo rendimiento.

Se realizará una estricta separación entre la parte algorítmica y gráfica, de forma que la parte algorítmica sea agnóstica de la parte gráfica y que esta última utilice la parte algorítmica cuando sea necesario. Cabe destacar que la entrada de eventos de usuario será implementada utilizando el mecanismo que ofrece el motor *Cocos2D-x* [10], de forma que no haya que preocuparse de realizar implementaciones separadas para *Microsoft Windows* [1] y para *Android* [2].

7.1 Algorítmica

En primer lugar vamos a analizar las decisiones tomadas para el diseño de las clases asociadas a la parte algorítmica del juego y haremos también un análisis detallado de la implementación algorítmica realizada.

7.1.1 Visión general

A continuación se enumeran las distintas clases asociadas a la parte lógica (en adelante I.A.) y se proporciona una breve descripción de las mismas:

- ***Card*** → Clase que define una carta de la baraja y también contiene distintas utilidades referentes al procesamiento de cartas.
- ***Player*** → Clase que define un jugador (si es humano o I.A., las cartas que tiene para jugar, las cartas que posee, etc.).
- ***Solver*** → Clase base del algoritmo de resolución del problema, proporciona la interfaz necesaria para ejecutar el algoritmo de resolución, bien sea sin conteo (estándar) o con conteo. Para ello, se le ha de proporcionar un *Player* y la lista de cartas que hay sobre la mesa.
    - ***StandardSolver*** → Especialización de la clase *Solver* para ejecutar el algoritmo sin conteo, únicamente teniendo en cuenta las cartas del jugador que le toca mover. En nuestro juego se utiliza cuando hay 3 o 4 jugadores.
    - ***CountSolver*** → Especialización de la clase *Solver* para ejecutar el algoritmo con conteo. *CountSolver* hace uso de *StandardSolver* para ejecutar el juego en cada ronda, realizando mini-max, es decir, asumiendo que el oponente siempre tiene las cartas con mayor valor añadido de las cartas que faltan por salir.
- ***AIManager*** → Proporciona la lógica global de una partida, controlando qué cartas hay sobre la mesa, qué cartas se reparten a los jugadores, el número de jugadores, si los movimientos proporcionados por los jugadores (humano o I.A.) son correctos, el modo de juego (estándar o asociación), el nivel de dificultad, etc.

7.1.2 Diagrama de clases

A continuación se presenta el diagrama *UML* [27] simplificado (únicamente listando los atributos y métodos más relevantes) del diseño de clases referente a la parte de I.A.:

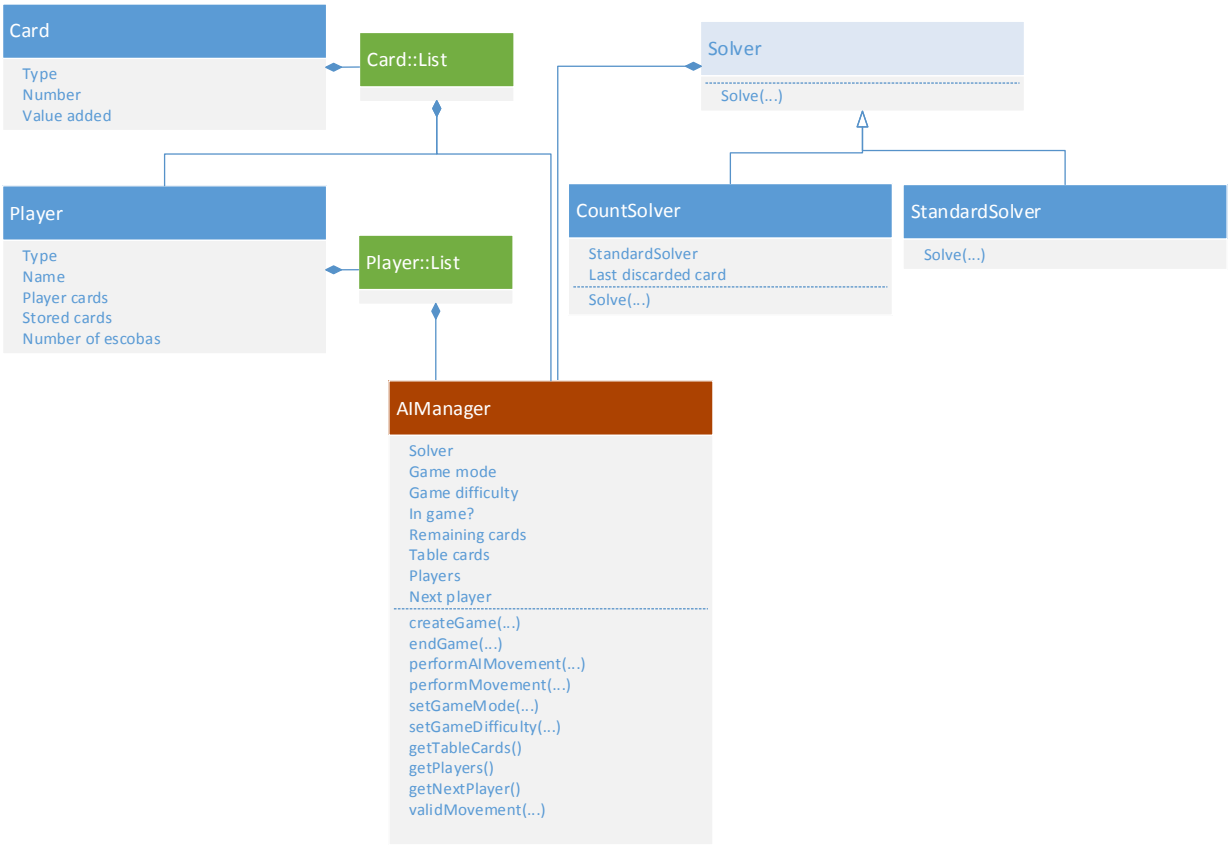


Figura 17 – Diagrama de clases de la parte algorítmica

7.1.3 Detalles de la implementación

A continuación vamos a ver en mayor detalle las partes más importante de la I.A.: los dos tipos de *Solver* y el *AIManager*.

7.1.3.1 Solver: StandardSolver y CountSolver

Como ya hemos introducido anteriormente, la parte responsable de ejecutar los algoritmos de resolución de problemas heredan de la clase base *Solver*.

La parte más relevante de dicha clase es la definición de la función *solve*, que han de implementar tanto *StandardSolver* como *CountSolver*:

```
virtual const Movement solve(const Card::List& table, Player* player) = 0;
```

Además, la clase *Solver* también almacena el modo de juego (estándar o asociación) así como el nivel de dificultad (fácil, medio o difícil), y estos se pasan a la misma a través de su constructor.

7.1.3.1.1 StandardSolver

StandardSolver es la especialización que se encarga de ejecutar el algoritmo sin conteo, explicado de forma teórica en la sección 2.4.2 - Algoritmo 3 o 4 jugadores: sin conteo.

Para ello, se realiza una expansión de nodos a partir de la situación inicial del jugador y la mesa. Esta expansión de nodos dará como resultado un árbol tanto con nodos válidos (cuya puntuación es 5, 15, 25,…) como con nodos inválidos (cuya puntuación es distinta de 5, 15, 25,…).

Una vez expandidos todos los posibles nodos, se realiza la búsqueda del mejor nodo válido, esto es, del nodo válido con mayor valor añadido. El valor añadido de un nodo se calcula como la suma de los valores añadidos de todas las cartas que intervienen en el movimiento y además se suma un punto extra si la jugada implica a todas las cartas que hay sobre la mesa, es decir, si hace escoba.

Para tener en cuenta la dificultad del juego, durante la búsqueda del mejor nodo se toma una mejor opción con distintas probabilidades, según la dificultad. Es decir, con una dificultad de juego media, y una vez hayamos encontrado el primer nodo válido, tomaríamos como mejor otro nodo que tuviese mayor valor añadido con una probabilidad del 65%. De esta manera no siempre escogemos el mejor movimiento y el jugador humano tiene más posibilidades de ganar.

Además, también hemos de tener en cuenta el modo de juego: estándar o asociación. Esto únicamente se tiene en cuenta a la hora de realizar la expansión de nodos.

Para el modo asociativo, los nodos tienen un *flag* que indica el tipo de nodo, que puede ser A o B (siguiendo con la nomenclatura utilizada en la sección 2.4.2.2 - Sistema de control (expansión del árbol)). De esta forma, los nodos de tipos A son aquellos que únicamente hay que expandir en sumas (sus hijos solo son nodos tipo A), y los nodos de tipo B son aquellos que hay que expandir en producto y suma (por lo tanto, generan una pareja de estados por cada combinación de cartas: uno de tipo A y otro de tipo B). Este *flag* es ignorado en el modo de juego estándar, ya que en este modo todos los nodos son de tipo A.

En el *StandardSolver* también se ha implementado un *timeout* para que la ejecución del algoritmo no dure más tiempo de un límite preestablecido. Para ello, se establecen dos valores: límite de tiempo permitido (en milisegundos), y número de expansiones (llamadas recursivas) entre las que se tiene que comprobar si dicho límite se ha superado. Esto se hace de así porque las llamadas al sistema operativo que se han de realizar (*clock_gettime* en *Android* [2] y *QueryPerformanceFrequency/QueryPerformanceCounter* en *Microsoft Windows* [1]) son relativamente costosas y, por tanto, es mejor expandir unos cuantos estados entre comprobación y comprobación en aras de no desperdiciar un valioso tiempo computacional comprobando el *timeout*. Los valores que se han seleccionado son: un límite temporal de $500ms$ para la ejecución del algoritmo y un valor de 50 expansiones por comprobación, es decir, comprobamos el *timeout* cada 50 expansiones de nodos. Si el *timeout* vence, se para la ejecución del algoritmo y únicamente se extraen las soluciones de los nodos que haya dado tiempo a expandir. Nótese que la implementación este *timeout* es muy importante para poder garantizar un juego fluido cuando hay muchas cartas sobre la mesa y para evitar, en estos casos, que el sistema operativo Android cierre la aplicación por no responder tras unos segundos.

Este *timeout* únicamente se implementa en el *StandardSolver* y no en el *CountSolver* ya que éste último hace uso del primero, como se estudia en detalle a continuación.

7.1.3.1.2 CountSolver

CountSolver es la clase que se encarga de realizar la I.A. aplicando conteo. Para ello, se expande un árbol de estados (nótese que no usamos la misma nomenclatura que para los nodos del *StandardSolver*, aquí se les llama estados) que representa las distintas posibilidades de movimientos (nodos del *StandardSolver* válidos) hasta que ambos jugadores se quedan sin cartas, recordemos que este modo únicamente se utiliza para 2 jugadores.

Por tanto, para expandir el árbol de estados, se hace uso de la clase *StandardSolver*, con el fin de obtener los distintos estados posibles y quedarnos con los válidos.

Una vez que el árbol de estados está expandido (cada hoja del árbol representa el último movimiento del reparte actual, ambos jugadores se han quedado sin cartas), se calcula cual es el estado de mayor valor añadido (para ello se suma en positivo el valor añadido de las jugadas del jugador máquina y con signo negativos las del jugador humano) y elegimos el movimiento que nos haría ir hacia ese estado. Todo esto está explicado detalladamente de forma teórica en la sección 2.4.3 - Algoritmo 2 jugadores: con conteo.

De forma análoga que en el *StandardSolver*, aquí también se utiliza un sistema de probabilidades a la hora de elegir la mejor solución para implementar los distintos niveles de dificultad. Esto también está explicado de forma teórica en la sección 2.4.3.

Cabe destacar que si el *CountSolver* detecta que el oponente ha dejado una carta sobre la mesa en la jugada anterior (se le notifica a través del *AIManager*) se elegirá, en caso de que sea posible, la jugada de mayor valor añadido que involucre a dicha carta, para de esta forma forzar al oponente a volver a dejar una carta sobre la mesa.

7.1.3.2 AIManager

Para concluir con la arquitectura de software de la parte de inteligencia artificial vamos a realizar un análisis del diseño y funcionamiento de su parte más importante: la clase *AIManager*.

La clase *AIManager* es el corazón de la lógica de juego, quien controla en todo momento el estado de la partida y se encarga de ir realizando las actualizaciones pertinentes para gestionar los movimientos de los jugadores tanto de tipo I.A. (movimientos calculador gracias al *solver*) como de tipo humano (movimientos proporcionado a través de la parte de interfaz gráfica, que es donde los introduce el jugador humano).

Internamente, el *AIManager* es una clase de máquina de estados donde se controla quién es el siguiente jugador y se almacenan todos los parámetros del juego (modo de juego, dificultad, etc). Al comenzar una nueva partida, el *AIManager* se pone en su estado inicial y registra tanto los distintos jugadores como el tipo de *solver* que se ha de utilizar (estándar para 3-4 jugadores y de conteo para 2 jugadores), además de repartir las cartas a los distintos jugadores y poner sobre la mesa 4 cartas, todo ello al azar.

El *AIManager* controla quién es el siguiente jugador que ha de mover y, en caso de que sea un jugador de tipo A.I., realiza de forma automática el movimiento cuando así se le ordena. Para el jugador humano, este movimiento se ha de introducir desde fuera y, previa comprobación de

que es válido, se ejecuta y se actualiza el estado del juego. Un movimiento puede ser tanto un movimiento de cuyo resultado el jugador obtiene cartas o un movimiento de descarte.

En partidas *1 vs 1*, el *AIManager* es también el encargado de notificar al *CountSolver* si el oponente ha dejado una carta sobre la mesa en la jugada anterior, para de esta forma intentar realizar un movimiento que involucre a dicha carta y obligar al contrincante a volver a dejar una carta sobre la mesa.

Además, el *AIManager* es el encargado de proporcionar un ganador cuando una partida acaba (o varios ganadores, en caso de empate). También controla todos los repartos de cartas cuando los jugadores se quedan sin cartas, es decir, gestiona desde el primer reparte hasta que se han repartido todas las cartas y se han realizado todas las jugadas hasta que los jugadores no tienen cartas y no quedan más cartas para repartir.

Finalmente, mencionar que el *AIManager* también realiza el control de rondas cuando se juegan torneos.

7.2 Parte gráfica

Una vez estudiada en detalle la implementación software de la parte algorítmica de la ESCOBA², nos centraremos en la arquitectura software utilizada para la parte gráfica de la aplicación – es decir, para la interfaz de usuario y la gestión de eventos de entrada -.

Gran parte de las decisiones de diseño software tomadas vienen motivadas por la arquitectura del motor *Cocos2D-x* [10], basada en escenas.

Por ello, en primer lugar vamos a realizar una introducción de la arquitectura del motor Cocos2D-*x* desde un punto de vista de usuario y posteriormente analizaremos la implementación realizada en las distintas pantallas del juego.

7.2.1 Visión general

Como hemos dejado entrever anteriormente, se va a organizar la arquitectura software de la aplicación tomando como unidad de referencia las escenas de Cocos2D-x, puesto que se va a utilizar este motor de juegos 2D.

De esta manera, cada menú y vista diseñados en la sección 5 de esta memoria, "Diseño del juego y sus distintos estados", será implementada como una escena de Cocos2D-x.

A continuación se incluye una detallada introducción al sistema de escenas y organización de elementos gráficos utilizados por el motor Cocos2D-x.

7.2.1.1 Escenas de Cocos2d-x

Las unidades lógicas superiores de una aplicación realizada con Cocos2D-x son las escenas.

Las escenas se implementan, generalmente, heredando de la clase *CCLayer*, implementando aquí la escena y finalmente creando un objeto de tipo *CCScene* al que se ancla el *layer*, que define todos los elementos de la vista así como la lógica de respuesta a los eventos de entrada.

A continuación se muestran dos fragmentos de código estándar con los que se crea y activa, respectivamente, una escena en Cocos2D-x:

```
// Esta función, estática, crea una escena con el menú principal
CCScene* MainMenu::scene()
{
	CCScene *scene = CCScene::create();
	MainMenu *layer = MainMenu::create();

	scene->addChild(layer);

	return scene;
}

// Esta porción de código crea una menú principal y lo activa, eliminando la
// escena que estuviese activa
CCScene* mainMenu = MainMenu::scene();
CCDirector::sharedDirector()->runWithScene(mainMenu);
```

Esta última porción de código se ejecuta cuando se pulsa un botón o se realiza una acción que nos lleva a una vista distinta.

Una escena de *Cocos2D-x* [10] sigue una estructura de árbol, formada por contenedores *CCLayer* y elementos que heredan de la clase *CCNode*, que son los distintos elementos gráficos con los que interacciona el usuario y a través de los cuales se muestra la información.

7.2.1.2 Elementos más comunes de Cocos2d-x

A continuación se describirán brevemente los elementos de Cocos2D-x utilizados en este proyecto. Estos elementos son los que se anclan a los contenedores de tipo *CCLayer*, ya introducido anteriormente, y que hereda de la clase *CCNode*.

Cabe destacar que también se puede utilizar un elemento de tipo *CCNode* como contenedor de otros elementos que heredan de *CCNode*. Esto es muy útil si queremos, por ejemplo, aplicar la misma transformación a un conjunto de elementos o si queremos mostrar u ocultar de una sola operación un conjunto de elementos gráficos (subclases de *CCNode*).

7.2.1.2.1 Layers

Como hemos mencionado anteriormente, las escenas en Cocos2D-x siguen una estructura de árbol teniendo, generalmente, un único contenedor raíz de clase *CCLayer* al que se anclan los distintos elementos que heredan de la clase *CCNode*.

En realidad, la clase *CCLayer* también hereda de la clase *CCNode*, implementando además la gestión de los distintos eventos de entrada, por ello se utiliza un nodo de tipo *CCLayer* como elemento raíz de la escena.

7.2.1.2.2 Menús y botones

Los elementos más importantes en una interfaz gráfica orientada a dispositivos táctiles son, sin duda, los botones.

En Cocos2D-x los botones se implementan a través de menús, objetos de tipo *CCMenu*, a los que se anclan los botones, objetos de tipo *CCMenuItemImage*.

Los objetos de tipo *CCMenuItemImage* permiten establecer dos texturas para los distintos estados del botón: normal y pulsado. Además, permiten definir *callbacks* que se invocan cuando estos botones son pulsados.

Una vez que se crean los elementos de tipo *CCMenuItemImage*, se establece su tamaño y posición dentro de la escena y se añaden al *layer* de la escena. Finalmente y para que el botón sea funcional (responda a los eventos de entrada) se ha de añadir a un objeto de tipo *CCMenu*.

Generalmente se añaden múltiples elementos de tipo *CCMenuItemImage* a un mismo objeto de tipo *CCMenu*.

7.2.1.2.3 Sprites

Otro de los elementos clave en cualquier interfaz 2D son los *sprites*, simples imágenes decorativas que suelen ser usadas a modo de fondo o para colocar elementos gráficos que no interaccionan con el usuario.

En Cocos2D-x los *sprites* se definen a través de objetos de la clase *CCSprite*. Estos elementos permiten definir una imagen y aplicar sobre ellos todas las transformaciones que se pueden realizar a la clase *CCNode*, de la cual heredan.

7.2.1.2.4 Labels

Finalmente, nos queda por presentar los *labels* de *Cocos2D-x* [10], que se definen a través de la clase *CCLabelTTF*.

Los *labels* son elementos gráficos que muestran un texto sin interaccionar con el usuario.

Los objetos de tipo *CCLabelTTF* nos permiten definir tanto la tipografía como el tamaño de fuente que ha de utilizar el *label*.

7.2.2 Detalles de la implementación

Una vez introducidos los distintos elementos con los que podemos crear escenas en el motor Cocos2D-x, estamos en condiciones de centrarnos en los detalles de la implementación realizada en cada una de las distintas pantallas del juego.

Como ya hemos mencionado en secciones anteriores, cada pantalla del juego se ha implementado mediante una escena de Cocos2D-x.

7.2.2.1 Menú principal

El menú principal se ha implementado haciendo uso de los siguientes elementos:

- *Sprites*:
  - Fondo.
  - Logo de la ESCOBA².
  - Texto de 'Jugador' para la selección de jugadores.
  - Espiral rotatoria para los retos.
- Menú con 6 botones:
  - 5 botones para las entradas principales del menú: nueva partida, nuevo torneo, estadísticas, tutorial y salir.
  - Un botón para consultar los retos.
- Menú con 5 botones, para la selección de los jugadores – implementados mediante botones que utilizan la misma textura, en escala de grises, pero que están tintados con distintos colores.

A continuación se muestra el resultado final del menú principal:

Figura 18 – Versión final del menú principal

7.2.2.2 Menú de configuración de la partida

El menú de configuración de la partida se ha implementado con los siguientes elementos:

- *Sprites* para los siguientes elementos:
  - Fondo.
  - Textos implementados mediante imágenes.
- Menú con 10 botones:
  - 6 botones estándar para comenzar el juego según la dificultad y el modo de juego.
  - 3 botones auto-exclusivos para seleccionar los jugadores, de tal forma que al seleccionar 2 jugadores los botones de 3 y 4 jugadores se desactiven.
  - Un botón para regresar al menú principal.

A continuación se muestra una captura del menú de configuración de la partida:

Figura 19 – Versión final del menú de configuración de la partida

7.2.2.3 Vista de juego

A continuación, nos centraremos en cómo se ha realizado la implementación gráfica de la vista de juego, sin duda alguna la pantalla más compleja con la que cuenta la ESCOBA².

Esta complejidad no se debe al número de elementos gráficos o los sub-estados con los que cuenta esta pantalla, sino que se debe a la lógica de interacción con el usuario que se ha debido implementar en este estado.

A continuación se explicará la implementación realizada para el elemento gráfico que representa las cartas y los distintos sub-estados de la vista de juego.

7.2.2.3.1 Cartas

En primer lugar nos centraremos en la implementación realizada para el elemento gráfico que representa una carta.

Una carta vendría a ser un botón pero con un conjunto de lógica adicional que modifica su estado (pulsable – *activo* - o no pulsable – *inactivo* -). Sin embargo, y a diferencia que en un botón tradicional de *Cocos2D-x* [10], no necesitamos dos imágenes para definir el botón, nos bastaría con una sola imagen cuya opacidad y tamaño cambiasen en función de si la carta está pulsada (mayor opacidad y tamaño) o seleccionada (mayor opacidad).

Además, sería muy útil que el propio objeto '*carta'* implementase cierta lógica de forma local para cambiar su propio estado en función de la partida: así, por ejemplo, una carta del tablero únicamente se activaría (para poder ser pulsada por el usuario) una vez que se hubiese seleccionado una carta perteneciente al jugador.

Por todas las razones anteriormente descritas, se ha decidido hacer una nueva clase para implementar el elemento gráfico que representa una carta: la clase *CardSprite* que hereda de *CCSprite* (funcionalidad de pintado) y *CCTargetedTouchDelegate* (respuesta a eventos de entrada).

Esta es la funcionalidad extra que se ha implementado en la propia carta:

- La clase *CardSprite* conoce cuál es la carta que representa y si pertenece a la mesa o al jugador.
- Cuando el comienzo de una pulsación (con el dedo / ratón) sobre la carta es detectada, su estado gráfico se altera (poniendo opacidad total y una escala $x1.2$) para notificar al usuario de que esa carta puede ser seleccionada únicamente cuando se dan las siguientes condiciones:
    - La carta no ha sido previamente seleccionada.
    - Pertenece al jugador y éste aún no ha seleccionado ninguna carta.
    - Pertenece a la mesa y el jugador ya ha seleccionado una de sus cartas previamente.
- Cuando una pulsación termina y se cumplen las condiciones listadas en el punto anterior (es decir, la carta es seleccionable por el usuario en ese momento del juego), se inyecta la carta pulsada a la actual jugada del usuario y se cambia su opacidad al 100% y su escala al tamaño original para notificar al usuario de que esa carta ya está seleccionada y no puede volver a utilizarse en la jugada actual.

7.2.2.3.2 Sub-estado de juego

Una vez detallada la implementación de las cartas, estamos en condiciones de explicar cómo se ha implementado el sub-estado principal de la vista de juego: el denominado sub-estado de juego, esto es, donde el usuario realiza su jugada: realizar una jugada mediante una combinación de cartas o pasar dejando una de sus cartas sobre la mesa.

Este sub-estado está integrado por los siguientes elementos:

- *Sprites*:
  - Fondo.
  - Durante el menú de confirmación de 'Rendirse', el fondo de dicho menú.
- Lista de cartas:
  - Lista de cartas sobre la mesa.
  - Lista de cartas del jugador, de 1 a 3 elementos - dependiendo de cuántas cartas haya utilizado hasta el siguiente reparte o fin de la partida.
- Botones:
  - *Reset*, jugar y pasar.
  - Rendirse.
  - Durante el menú de confirmación de 'Rendirse' hay dos botones más: sí y no.
  - En el modo de juego asociativo: botón '+', para pasar de multiplicar a sumar cartas.
- *Labels*:
  - Reparte actual – indica en qué reparte nos encontramos del total de reparte que tiene la partida.
  - Puntuación de la jugada actual – además, su color varía en función de si la jugada es válida (color verde) o no (color blanco).
  - Durante un torneo, la ronda actual.

Estos elementos se actualizan (muestran, esconden, inactivan, etc.) en función del estado actual del juego.

Como ya hemos mencionado anteriormente, el diseño de la aplicación se ha realizado manteniendo una estricta separación entre la parte gráfica (escenas de *Cocos2D-x* [10]) y la parte algorítmica (*AIManager*). Durante el sub-estado de juego, la jugada se va almacenando en la parte gráfica y finalmente se inyecta en el *AIManager* para que se realice la jugada.

Además, cada vez que el jugador modifica la jugada se realiza un chequeo rápido desde la parte gráfica (utilizando utilidades de la parte algorítmica) para comprobar si la jugada actual es válida o no (es decir, si es un número entero cuya cifra menos significativa es un 5: 5, 15, 25, 35…) y actualizar el color del *label* de la jugada (en dificultad fácil y media, donde se muestre este *label*) así como activar el botón de '*Jugar'*.

7.2.2.3.3 Sub-estado de visualización de las jugadas del turno actual

Una vez que el usuario ha realizado una jugada (bien sea con una jugada válida o desechando una carta y depositándola sobre la mesa), se muestra un sub-estado que muestra tanto la jugada que ha realizado el usuario como las jugadas que han realizado el resto de jugadores – controlados por el *AIManager* -.

Este sub-estado se implementa de la siguiente manera:

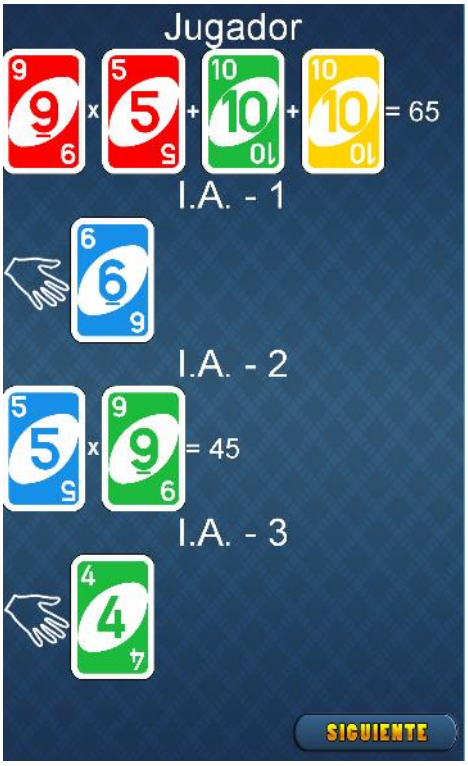
1. Se establece un fondo (s*prite*) por encima del sub-estado de juego que pasa de transparente a opaco en 0.65 segundos.
2. Cada 0.25 segundos se añade la jugada de uno de los jugadores a la pantalla, en orden y empezando siempre por el jugador humano. En este paso, cada 0.25 segundos y por cada adversario, se ordena al *AIManager* que ejecute la jugada del siguiente jugador manejado por la inteligencia artificial y muestre el resultado en una transición de 0.25 segundos. Esto se hace así para minimizar la sensación de bloqueo mientras se ejecuta el algoritmo de resolución: en lugar de calcular todas las jugadas de los contrincantes al mismo tiempo, lo hacemos cada 0.25 y mostrando animaciones y transiciones entre medias – de esta forma el usuario disfruta de una experiencia de juego fluida y satisfactoria.
3. Una vez que se han calculado y mostrado todas las jugadas del turno actual, se muestra un botón de siguiente para que el usuario pueda volver al sub-estado de juego y jugar el siguiente turno.

Como se ha realizado para el resto de estados y sub-estados, a continuación se incluye una lista de los distintos elementos gráficos con los que se implementa este sub-estado:

- *Sprites*:
    - Fondo.
    - Para las cartas.
    - Para los operadores y símbolos matemáticos: ‘x’, ‘+’, ‘(‘, ‘)’.
    - Para la imagen de desechar una carta.
- *Labels*:
    - Para el nombre de los jugadores.
- Botones:
    - Únicamente el botón de siguiente.

Finalmente incluimos una captura de pantalla de los dos sub-estados de la pantalla de juego:

1. Sub-estado de juego



2. Sub-estado de visualización de las jugadas del turno actual.

Figura 20 – Versión final de los sub-estados de la pantalla de juego

7.2.2.4 Vista de resultados

Finalmente y tras jugar todos los turnos de la partida, se muestra el resultado de la partida actual, desglosándose los puntos de cada jugador además de indicar qué jugador ha ganado y un mensaje que varía en función de si el jugador gana (se le da la enhorabuena), pierde (se muestra un mensaje de consolación) o se produce un empate. Este mensaje también varía si estamos jugando un torneo, en este caso muestra "Ronda Finalizada".

Esta pantalla de juego se implementa utilizando los siguientes elementos:

- *Sprites*:
  - Fondo.
  - Logo de la ESCOBA².
  - Imagen que indica si se ha ganado, perdido o empatado.
  - Fondo de la tabla de resultados.

- *Labels*:
  - Puntos de cada jugador. Un *label* para la puntuación de cada jugador (de 2 a 4 *labels*, en función del número de jugadores).
  - Nombre del jugador que ha ganado.
- Botones:
  - Botón de "*Entendido*".

A continuación se muestra una captura de la pantalla de resultados dentro de la vista de juego para las 3 opciones posibles: el jugador humano gana, pierde o se produce un empate:

1. El jugador gana

2. El jugador pierde

3. Se produce un empate

Figura 21 – Versión final de la pantalla de resultados

7.2.2.5 Menú de configuración de torneos

Con una implementación similar a la del menú de configuración de la partida, el menú de configuración de torneos está integrado por los siguientes elementos:

- *Sprites* para:
  - Fondo.
  - Textos implementados mediante imágenes.
- *Label* para mostrar el número de rondas seleccionadas.
- Menú con 4 botones:
  - Subir y bajar el número de rondas del torneo.
  - Comenzar el torneo.
  - Volver al menú principal.

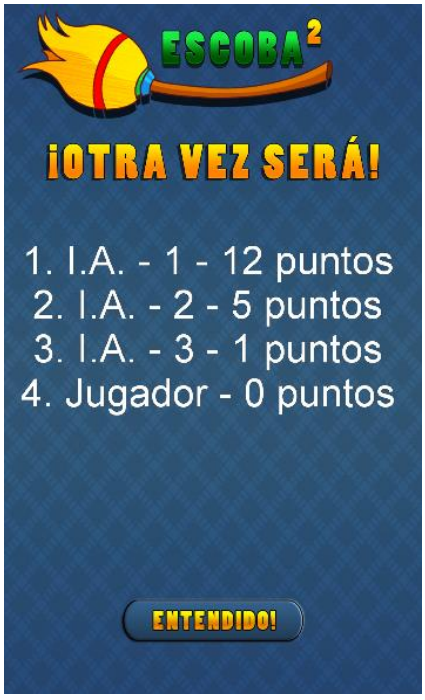
Además, en modo torneo se muestra una pantalla de resultados tras finalizar todas las rondas donde se muestra la clasificación final del torneo ordenando a los jugadores por puntuación acumulada y un mensaje que varía en función de si el usuario pierde, gana o se produce un empate.

Esta pantalla está integrada por los siguientes elementos:

- *Sprites* para:
  - Fondo y logo.
  - Textos implementados mediante imágenes.
- *Label* para mostrar la clasificación.
- Botón para volver al menú principal.

Se incluye a continuación tanto una captura del menú de configuración de torneos como una captura de la pantalla de clasificación final del torneo:

1. Menú de configuración de torneos



2. Pantalla de clasificación final del torneo

Figura 22 – Versión final de las vistas asociadas al modo torneo

7.2.2.6 Estadísticas

La pantalla de estadísticas se ha implementado utilizando los elementos citados a continuación:

- *Sprites* para:
  - Fondo.
  - Textos implementados mediante texturas (estilizados).
  - Jugadores – implementados mediante *sprites* que utilizan la misma textura, en escala de grises, pero tintados con distintos colores.
- *Labels* para cada uno para cada uno de los elemento de la matriz 5x3 de puntuaciones.
- Un botón para regresar al menú principal.

Este es el resultado final de la pantalla de estadísticas:

Figura 23 – Versión final de la pantalla de estadísticas

7.2.2.7 Ayuda

Finalmente, la pantalla de ayuda se ha implementado utilizando los siguientes elementos:

- *Sprites* para:
  - Fondo.
  - 2 imágenes de ayuda que se deslizan de abajo hacia arriba (dos en lugar de una ya que el máximo tamaño de una textura en *OpenGL ES 2.0* [5] es $2048x2048$, insuficiente para el texto de ayuda). Está implementada mediante dos *sprites* a los que se modifica su posición en cada *frame*. Además, se detecta cuando el usuario arrastra su dedo por la pantalla para parar el deslizamiento y mover los *sprites* según la posición del dedo del usuario.
- Botón para volver al menú principal.

Finalmente se incluyen dos capturas de pantalla de la implementación de la pantalla de ayuda:

Figura 24 – Versión final de la pantalla de ayuda

7.3 Datos de usuario

Uno de los aspectos que se ha debido de tratar con especial cuidado es el almacenamiento de datos dinámico por parte de la aplicación.

Esto se debe a dos razones:

1. Cómo se guardan los datos generados por aplicaciones en *Android* [2] (no en espacio de la aplicación sino en la tarjeta de memoria o la caché).
2. El requisito de no necesitar permisos especiales, lo que también incluye no necesitar permiso para escribir datos en el dispositivo de almacenamiento externo [28].

Nota: por almacenamiento de datos dinámico nos referimos a los datos que altera la aplicación entre ejecuciones, es decir, datos de usuario.

En este proyecto, la ESCOBA², únicamente hemos de guardar dinámicamente las estadísticas de los 5 jugadores disponibles en el juego.

7.3.1 Formato de los datos

El sistema de estadísticas de la ESCOBA² almacena, para cada jugador, los siguientes datos:

- Nombre del jugador → *String* (cadena de caracteres)
- Color *RGB* identificativo del jugador → 3 números enteros
- Juego simples empezados, perdidos y ganados. → 3 números enteros
- Torneos empezados, perdidos y ganados. → 3 números enteros

Por tanto, la forma más directa de guardar y leer estos datos de forma dinámica y que estos estén disponibles entre distintas ejecuciones del programa es utilizando un fichero de texto.

A continuación se propone el siguiente formato de almacenamiento:

NOMBRE_JUGADOR ***R G B***
juegos_simples_empezados juegos_simples_ganados juegos_simples_perdidos
torneos_empezados torneos_ganados torneos_perdidos **\n**

Como podemos observar, en el fichero de texto se asigna una línea (hasta el carácter **\n**) a cada jugador. A continuación se muestra un ejemplo para el jugador rojo cuando todavía no ha jugado ninguna partida:

Rojo 255 0 0 0 0 0 0 0 0 \n

Una vez que ya hemos propuesto una estructura de fichero válida para almacenar las estadísticas estudiaremos las distintas alternativas de implementación disponibles.

7.3.2 Alternativas para la implementación

A continuación se estudiarán las distintas alternativas disponibles para la implementación del almacenamiento de estadísticas teniendo en cuenta las limitaciones existentes en Android.

Se proponen dos alternativas:

1. Utilizar ficheros de texto.
2. Utilizar la clase *CCDictionary* de *Cocos2D-x* [10].

7.3.2.1 Utilizar ficheros de texto

En primer lugar vamos a estudiar la opción de guardar las estadísticas en ficheros de texto que son generados desde el código C++ de la aplicación y leídos posteriormente también desde el código C++ de la aplicación.

Esta opción utilizaría llamadas nativas a *fopen/fclose* [29] para crear, leer, editar y guardar el fichero de estadísticas.

Es la opción más sencilla, pero existe un problema en *Android* [2]: necesitamos el directorio de la caché de la aplicación que se ha de obtenerse a través del contexto de la aplicación [30], en la parte *Java* [9].

Otra opción sería guardarlo en la tarjeta de memoria (ruta */sdcard* en dispositivos Android), pero para ello se requiere un permiso especial: acceso a la tarjeta de memoria, a través del permiso *WRITE_EXTERNAL_STORAGE* [28]. No obstante esto no es una opción pues es requisito indispensable el no utilizar permisos extra en la aplicación (ver requisitos en la sección 1.2.2 – Requisitos técnicos).

Por tanto, si se quiere implementar esta opción hemos de utilizar código no multiplataforma para obtener el directorio de la caché de la aplicación a través del *JNI (Java Native Interface)* [31]. Es por ello que vamos a explorar una segunda opción que nos permita utilizar código íntegramente multiplataforma.

7.3.2.2 Utilizar la herramienta CCDictionary de cocos2d-x

El motor *Cocos2D-x* [10] trae integrado un sistema tipo diccionario, que almacena pares llave/valor, implementado a través de la clase *CCDictionary* [32].

La gran ventaja de esta alternativa es que Cocos2D-x implementa toda la parte no multiplataforma de forma interna y, por tanto, el API de *CCDictionary* es totalmente multiplataforma.

La desventaja de esta opción es su velocidad. Dependiendo del sistema operativo, se utiliza una implementación distinta: almacenamiento en formato *xml* [33], usando bases de datos en Android, utilizando *NSDictionary* [34] en *iOS* [17], etc. No obstante, esta implementación será siempre más lenta que si utilizamos un fichero de texto de forma directa, donde los datos se leen directamente y no es necesario procesar ningún tipo de estructura adicional.

7.3.3 Alternativa seleccionada

Dando prioridad al código multiplataforma y puesto que el volumen de datos a guardar para las estadísticas es pequeño – y por tanto no se necesita un altísimo rendimiento – se ha optado por utilizar la clase *CCDictionary* de Cocos2D-x, es decir, la segunda opción de las estudiadas anteriormente.

En este caso, almacenamos pares llave/valor en un diccionario, ambos son cadenas de caracteres:

- Llave: el nombre del jugador, ejemplo: "*Rojo*".
- Valor: color RGB del jugador; juego simples empezados, perdidos y ganados; torneos empezados, perdidos y ganados. Ejemplo: "*255 0 0 0 0 0 0 0 0* ".

7.3.4 Funcionamiento de las estadísticas

Finalmente se explicará el funcionamiento de las estadísticas en la ESCOBA², esto es, cómo se generan las estadísticas en la primera ejecución del programa y en qué puntos se actualiza o se lee el fichero de estadísticas.

En primer lugar, cabe destacar que dentro de la arquitectura software del juego las estadísticas están manejada por la clase *UserManager*. Esta clase maneja todo lo referente al jugador seleccionado por el usuario – recordemos que el usuario puede elegir entre cinco jugadores distintos -, desde qué jugador tiene seleccionado el usuario hasta cuántas partidas se han iniciado y cuantas se han llegado a perder o a ganar.

Cuando la aplicación arranca, la clase *UserManager* intenta cargar los datos a través de un objeto *CCDictionary* [32] que hace referencia al archivo '*users.plist*. Paso previo comprueba si dicho archivo existe, y en caso contrario significa que es la primera ejecución (o bien que el usuario ha eliminado de forma intencionada la caché de la aplicación).

En caso de que el archivo no exista, se generan los usuarios/estadísticas iniciales y se guardan en el archivo '*users.plist'*. En caso contrario, si el archivo existe, se leen los pares llave/valor y se extraen los valores guardados de la cadena de caracteres almacenada.

Cuando el usuario, que ha seleccionado un jugador en el menú principal – pudiendo utilizar el que hay seleccionado por defecto - inicia un juego simple o un torneo, se notifica a la clase *UserManager* y ésta actualiza y guarda las estadísticas. Así mismo, cuando el usuario gana o pierde una partida simple o un torneo, también se notifica al *UserManager* y se vuelven a guardar las estadísticas en disco.

Si el usuario abandona una partida el contador de juegos simples o torneos empezados, según sea el caso, se incrementa pero nunca se llega a ganar o a perder. Es por ello que la suma de los juegos ganados y perdidos siempre será igual o menor que el número de juegos empezados.

Restando el número de juegos ganados o perdidos al número de juegos empezados podemos descubrir el número de juegos que el usuario ha abandonado.

7.4 Soporte de temas

Como hemos visto anteriormente, el juego ESCOBA² integra un sistema de retos para desbloquear barajas de cartas conforme el número de partidas ganas por un jugador se vaya incrementando, motivando de esta forma al usuario.

Internamente, el sistema de retos se implementa a través de una clase, *UIManager – User Interface Manager* -, al que se le piden todas las rutas de las texturas que se utilizan a lo largo del juego.

Esta clase, únicamente tiene dos métodos:

- void setTheme(const std::string& n, const std::string& p);
- `const std::string getResourcePath(const std::string& file) const;`

A través del primer método, se establece el tema actual. A través del segundo método se pide la ruta para una imagen determinada.

El tema por defecto está en la carpeta '*theme_default'*, dentro de esta carpeta se incluye la versión por defecto de todas las imágenes del juego. A través de *setTheme(...)* nosotros podemos establecer otro tema, por ejemplo: *setTheme("Baraja española", "theme_spain").*

De esta manera, cuando nosotros pedimos la ruta de una imagen al *UIManager* (por ejemplo: "*1_oros.png*"), primero se comprueba si existe una versión con el tema usado (*theme_spain* en este ejemplo) y si es así se devuelve la ruta "*theme_spain/1_oros.png*". En caso contrario, si el tema actual no dispone de una versión personalizada de la textura que nos piden, se devuelve la textura por defecto: "*theme_default/1_oros.png*".

De esta manera, podemos aplicar distintos temas visuales al juego en tiempo real, de forma dinámica, pues cuando se crea una escena se vuelven a pedir las rutas para los distintos recursos.

Cuando un reto se desbloquea y tenemos acceso a una baraja distinta, nosotros establecemos un tema distinto donde únicamente se definen las texturas de la baraja. De esta forma la baraja luce de una forma diferente mientras que el resto de elementos se mantienen como antes, pues se utilizan los recursos de *theme_default* para todo lo que no esté disponible en el tema establecido.

7.5 Gestión de múltiples resoluciones

El juego ESCOBA² está diseñado para ofrecer una buena calidad visual en un amplio rango de resoluciones: desde $240x320$ pixeles hasta $1440x2368$ pixeles.

Para ello, se han tenido en cuenta una serie de requisitos desde el principio de la etapa de diseño gráfico e implementación:

- Se debe de ver con máxima calidad a la máxima resolución soportada: $1440x2368$.
- El juego ha de verse correctamente, aunque se acepta algo de *aliasing* [35], en la mínima resolución soportada: $240x320$.
- El juego ha de verse correctamente para todas las resoluciones intermedias, siempre y cuando la relación de aspecto sea mayor que $1:2$ y menor que $2:3$.

Para cumplir con los requisitos de resolución visual, se han impuesto las siguientes restricciones durante todo el desarrollo del proyecto:

- La resolución nativa de la interfaz de usuario es igual a la máxima resolución soportada: $1440x2368$ pixeles. Por tanto todos los elementos visuales se han diseñado a esta resolución.
- Dado que los monitores de *PC* en los que se ha desarrollado el proyecto tienen una resolución inferior a la máxima resolución soportada, los contenidos de la ventana del juego se han escalado por un factor 0.5 durante todo el desarrollo.
- Para el resto de resoluciones soportadas, se utiliza *filtrado bilineal* [36] con el objetivo de suavizar el escalado de las texturas.

7.6 Resumen arquitectura software

Para finalizar la sección de arquitectura software, realizaremos un pequeño resumen desde un punto de vista global y sin entrar en los detalles de implementación de la arquitectura software utilizada, pero que resulta muy útil para dar cohesión a todas las partes estudiadas anteriormente en esta misma memoria.

Como ya hemos mencionado anteriormente, la arquitectura software se ha realizado dividiendo la implementación en dos partes claramente diferenciadas: parte algorítmica y parte gráfica.

A continuación se incluye una tabla con las distintas clases que componen cada una de las dos partes:

Algorítmica	Gráficos
AIManager	AppDelegate
Solver	MainMenu
StandardSolver	SelectGameMenu
CountSolver	GameView
Player	NewTournamentMenu
Card	ResultView
	StatsView
	TutorialView
	UIManager
	UserManager

Tabla 4 – Clases de la arquitectura software

La parte algorítmica ya ha sido minuciosamente estudiada en la sección 7.1 – Arquitectura software | Algorítmica.

Así mismo, la parte gráfica también ha sido explicada en detalle en la sección 7.2 – Arquitectura software | Parte gráfica.

A continuación, lo que haremos será resaltar los puntos más importantes de interacción entre ambas partes. Este análisis únicamente se realizará en un sentido: *gráficos* → *algorítmica*, ya que la parte algorítmica es totalmente agnóstica de la parte gráfica.

La clase que conecta la parte gráfica con la algorítmica es *AIManager*, que se usa en las siguientes clases de la parte gráfica:

- *SelectGameMenu* → Es la clase que implementa la pantalla de configuración de la partida, extudiada en la sección 7.2.2.2 - Menú de configuración de la partida. Esta clase utiliza el AIManager para iniciar la partida en función de las opciones seleccionadas por el usuario (modo de juego, dificultad y número de jugadores).
- *GameView* → La clase *GameView* implementa la vista de juego, cuya implementación ha sido estudiada en la sección 7.2.2.3 - Vista de juego. Como cabría pensar, la vista de juego hace un extenso uso de la clase *UIManager* ya que es la vista encargada de actualizar el progreso de la partida, para lo que inyecta las jugadas del usuario además en el *UIManager* además de indicarle cuando ha de ejecutar el algoritmo de resolución de juego para realizar los movimientos de los contrincantes.

7.6.1 Diagrama UML

A continuación se incluye un diagrama *UML* [27] de la arquitectura software completa, incluyendo tanto la parte algorítmica como la parte gráfica:

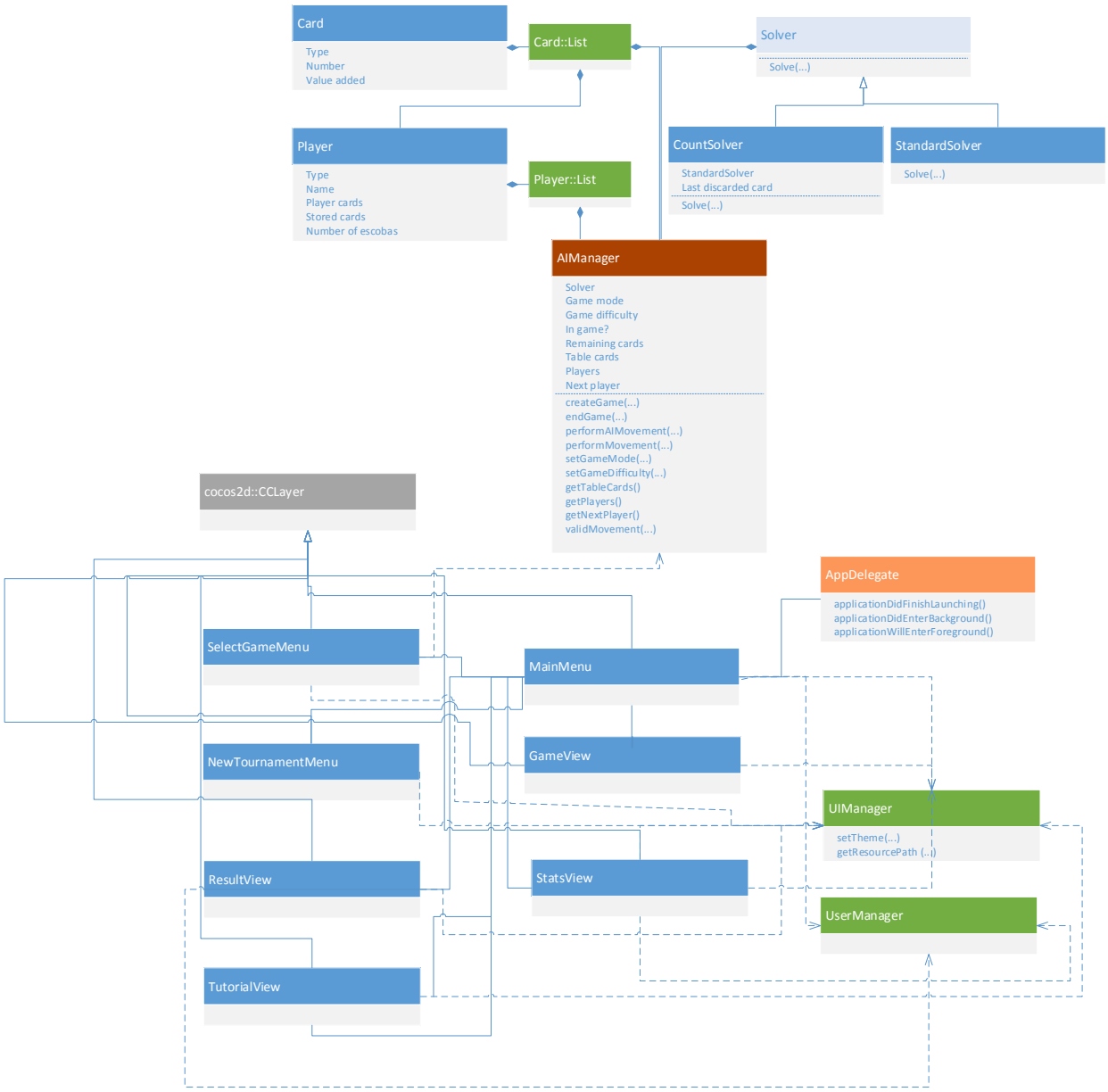


Figura 25 – Diagrama UML simplificado de la arquitectura software completa

Nota: el diagrama UML se ha simplificado con el objetivo de transmitir el diseño software desde un punto de vista global sin desviar el foco de atención a los pequeños detalles, ya estudiados anteriormente.

Como podemos observar, el único punto de unión funcional entre la parte gráfica (parte inferior del diagrama) y la algorítmica (parte superior) es el *AIManager*. Destacamos el matiz 'funcional' ya que en ciertos sitios de la parte gráfica se han utilizado estructuras de datos de la parte algorítmica, como por ejemplo las clases *Card* y *Player*.

8 Optimización

Llegados a este punto se discutirá sobre las decisiones que se han tomado durante la implementación de la aplicación en vista a la optimización de la misma, atendiendo a tres aspectos:

1. **Ciclos de reloj**: complejidad de las operaciones que se realizan de forma masiva, en nuestro caso las realizadas para encontrar las jugadas válidas por parte de la I.A.
2. **Acceso a memoria**: hoy en día el acceso a memoria es una operación más lenta que las operaciones aritméticas realizadas por la *ALU/FPU* [37], por tanto hemos de prestar especial atención a cómo se consume la memoria durante el procesamiento masivo de datos.
3. **Tamaño de la aplicación**: es requisito indispensable desde el inicio del proyecto que el tamaño final de la aplicación *Android* [2] sea inferior a $10MB$. Se estudiarán todos los procesos realizados para haber conseguido un tamaño inferior a $6'2MB$.

8.1 Ciclos de reloj y acceso a memoria

En esta sección se van a explicar las distintas decisiones de diseño que se han tomado con el fin de que el juego sea fluido, tanto desde un punto de vista de fotogramas por segundo - durante las transiciones y animaciones del juego - como a la hora de ejecutar el algoritmo de resolución del problema, que debido a la explosión combinatoria podría llegar a ser del orden de segundos en determinados casos.

En primer lugar, cabe destacar que la decisión más importante que se ha tomado para conseguir un buen rendimiento es la de utilizar código nativo, $C++$ en este caso (a través del *NDK* [16]), en lugar de utilizar código *Java* [9], que recordemos es código interpretado y por tanto más lento [38] [39], lo que cobra especial importancia en aplicaciones donde un buen rendimiento es crítico, como es el caso que nos ocupa.

Además de utilizar código nativo, hay ciertas prácticas que proporcionan una mejora de rendimiento en algoritmos iterativos y que manejan pequeñas cantidades de memoria de forma repetitiva, especialmente en determinadas arquitecturas *ARM* (dependiendo de si son *in-order* o *out-of-order* [40]):

- Evitar alocar de forma dinámica pequeños trozos de memoria. Es mejor reservar espacios relativamente grandes e ir usándolo conforme lo necesitemos.
- Acceder a posiciones de memoria que quepan en la misma línea de caché del procesador, con el objetivo de no tener que transferir de estas secciones de memoria desde otras cachés o incluso desde memoria principal.
- Evitar utilizar bloques condicionales que dependan de una posición de memoria que produzca *cache misses* [7] pues conlleva esperas, especialmente en procesadores *in-order* [40]. En ocasiones y si es posible, es mejor realizar más operaciones que comprobar condicionalmente si algo se ha de realizar pues transferir entre memorias es mucho más lento que ejecutar instrucciones sobre memoria que ya está en la línea de caché del procesador.
- No acceder a disco ni a recursos de red en operaciones iterativas.

Todas estas pautas han sido seguidas, en la medida de lo posible, durante el proceso de implementación de este proyecto.

8.2 Tamaño de la aplicación

Uno de los aspectos más importantes en cualquier aplicación destinada al ecosistema móvil – ya sea un juego o una aplicación de propósito general – es su tamaño. Los dispositivos móviles cuentan generalmente con un espacio bastante limitado y las conexiones $3G$ a través de las cuales los usuarios se suelen descargar las aplicaciones son limitadas, tanto en velocidad como en volumen de datos.

En un juego donde todos los elementos son imágenes esto cobra especial relevancia, especialmente si se soportan resoluciones *HD* (de hasta $2048x1024$), como es el caso que nos ocupa.

En este proyecto, la gran mayoría de elementos gráficos (imágenes / texturas) superan en resolución $512x512$ pixeles, utilizando 4 canales por pixel (R, G, B y A). Sin compresión y asumiendo 1 byte por componente, tendríamos que una única textura de $512x512$ ocuparía:

$$tamaño = (512x512)x4x8 = 8388608\ bits = 1048576\ bytes \approx 1MB$$

En el proyecto que nos ocupa tenemos más de 160 texturas, algunas llegando incluso a resoluciones de $1200x1400$ pixeles. Por tanto, si no usásemos formatos comprimidos podríamos estar hablando de un juego 2D para móvil con un tamaño en el rango de los $150 - 200\ MB$, algo totalmente inaceptable para este tipo de aplicaciones.

8.2.1 Formato PNG

Dentro de los numerosos formatos de imagen que hay disponibles, se ha seleccionado el formato *PNG* [41] para codificar las texturas debido, principalmente, a las siguientes características:

- Soporta transparencias.
- Compresión sin pérdidas.
- Altos ratios de compresión.
- Soportado por *Cocos2D-x* [10].
- Gran soporte por parte de las principales herramientas de edición de gráficos.
- Uso libre.
- Soporte de imágenes indexadas (Ver siguiente sección).
- Soporte de *dithering* [42].

8.2.2 Cuantización de texturas PNG

Pese al excelente ratio de compresión ofrecido por PNG en su versión de compresión sin pérdidas, el tamaño de la aplicación seguía siendo demasiado elevado – $25\ MB$ de texturas.

Por tanto, se ha procedido a la cuantización de las texturas mediante dos técnicas soportadas por el formato PNG:

- Indexación: usar paletas de colores, de $2^n, con\ n = 1,2,3,4,5,6,7,8$.
- *Dithering* [42]: añadir ruido de forma intencionada para reducir la percepción de los errores de cuantización.

Para realizar la cuantización de las texturas existentes de una manera rápida, se ha utilizado la aplicación *pngquant* [43] a través de la interfaz de usuario *PNGoo* [44].

PNGoo [44] nos permite seleccionar una lista de imágenes *PNG* [41] y ejecutar, de forma automática, la cuantización y el *dithering* [42] sobre las mismas, sobrescribiendo las imágenes PNG al finalizar el proceso.

A continuación se muestran dos capturas de pantalla de la aplicación *PNGoo*:

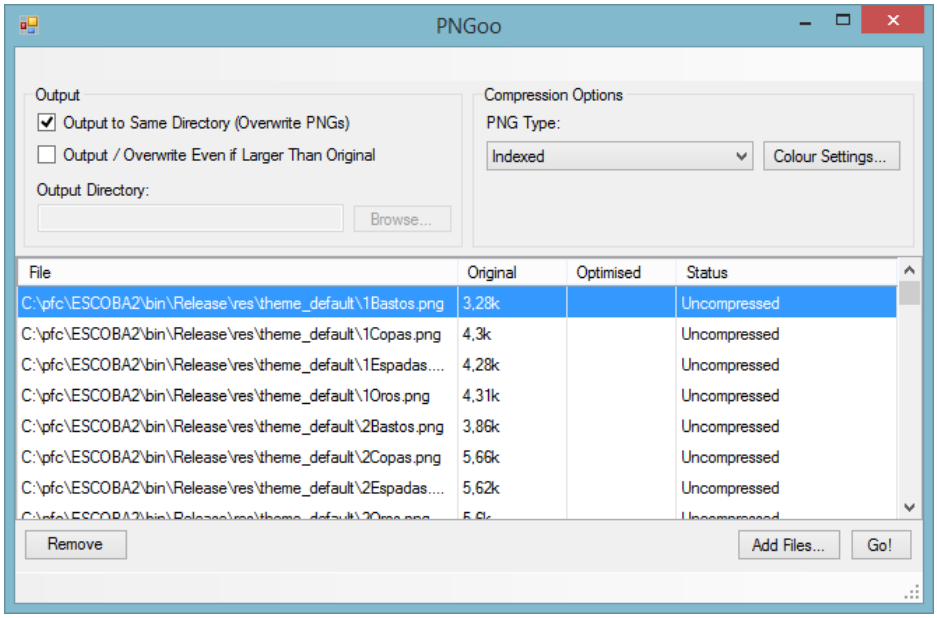


Figura 26 – Vista principal de la aplicación PNGoo

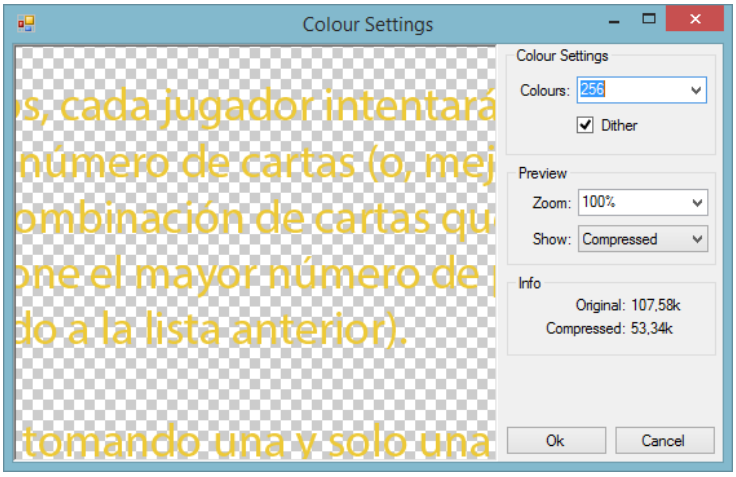


Figura 27 – Selección de opciones de cuantización de la aplicación PNGoo

En la gran mayoría de imágenes se ha utilizado una paleta de 256 colores combinado con *dithering*, obteniendo un tamaño final de la aplicación inferior a 6.2 *MB* en *Android* [2], un resultado más que satisfactorio teniendo en cuenta el elevado número de texturas con las que cuenta la ESCOBA².

Cabe destacar que la diferencia de calidad entre las texturas antes y después de la cuantización no es perceptible.

9 Conclusiones

Llegado a este punto y como apartado final de esta memoria, a continuación se incluyen la planificación y presupuesto, conclusiones finales y futuras líneas de trabajo derivadas del presente proyecto fin de carrera.

9.1 Planificación y presupuesto

En este apartado se incluye la planificación y el presupuesto, que se han realizado una vez concluida la etapa de investigación del proyecto – ya que es entonces cuando estamos en condiciones de planificar la etapa de implementación y elaborar un presupuesto estimado de los costes totales del proyecto.

9.1.1 Planificación

Este proyecto será realizado en un tiempo estimado de 10 meses, tiempo en el que también se ha incluido la duración de la etapa de investigación tratada anteriormente en esta misma memoria.

La fase de diseño del proyecto ha dado comienzo en noviembre de 2013 y se estima que el proyecto quedará finalizado en septiembre de 2014.

Se trabajará en el proyecto un tiempo medio de 3 horas al día durante los días laborables del mes. Cogiendo como referencia el valor medio de 22 días laborables al mes, esto nos da como resultado un total de 220 días de duración y un tiempo acumulado de desarrollo de 660 horas.

A continuación se muestra un diagrama de *Gantt* donde se muestra la planificación temporal de las distintas etapas del proyecto:

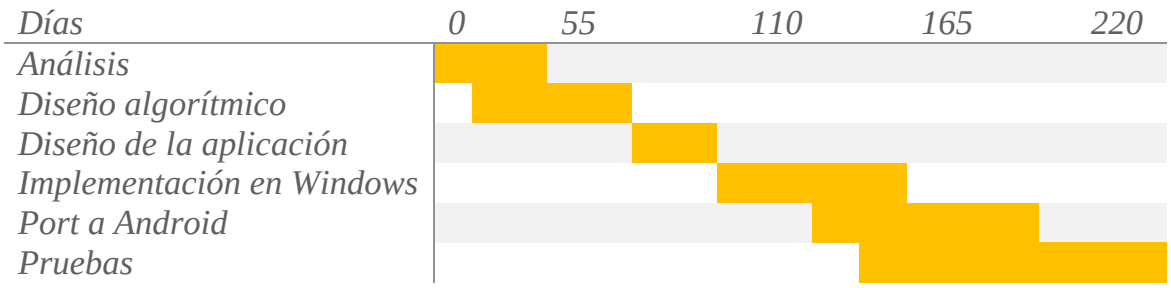


Figura 28 – Diagrama de Gantt

Como podemos observar en el diagrama de Gantt de la figura 4, el desarrollo se ha divido en 6 partes bien diferenciadas listadas por orden cronológico de realización.

Así mismo, podemos observar como ciertas partes se han solapado ya que la conclusión de ciertas etapas requería de un trabajo inicial en la etapa siguiente.

9.1.2 Presupuesto

A continuación se incluye el presupuesto estimado del proyecto. En dicha estimación, se ha tenido en cuenta tanto el salario de las personas implicadas en el desarrollo (tutor y autor en este caso), como el material y las licencias software necesarias para llevar a cabo el proyecto.

Costes directos

	Personal	**Categoría**	**Tiempo**	**Coste hora**		**Coste total**
		Ingeniero	660 horas	35€		23.110€
		Doctor	90 horas	60€		5.400€
	Equipos	**Descripción**	**Coste (IVA incl.)**	**Dedicación**	**Periodo de depreciación**	**Coste imputable**
		Acer Aspire V3-571G	899€ (21% IVA)	10 meses	60 meses	123,82€
		Galaxy Nexus	599€ (21% IVA)	2 meses	36 meses	27,5€
	Licencias	**Descripción**	**Coste (IVA incl.)**	**Dedicación**	**Periodo de depreciación**	**Coste imputable**
		Microsoft Word 2013	135€	10 meses	60 meses	18,6€
		Photoshop CS6	940€	8 meses	60 meses	103,58€
	Total CD					**28.773,5**
Costes indirectos						
		Estimación 20% CD				5.754,7€
	Total CI					**5.754,7€**
Total						**34.528,2€**

Tabla 5 – Presupuesto del proyecto

Como podemos observar, el presupuesto total del proyecto asciende a 34.528,2€.

9.2 Objetivos logrados

En primer lugar, cabe destacar que el mayor y más importante objetivo logrado es haber creado el juego ESCOBA² para las plataformas *Microsoft Windows* [1] y *Android* [2].

En la fase de diseño de la ESCOBA², nos encontrábamos con una serie de requisitos tanto técnicos como funcionales (ver sección 1.2 – Requisitos) que nos llevaron a tomar una serie de decisiones durante la etapa de diseño que posteriormente se vieron reflejadas y materializadas en la etapa de implementación.

Llegados a este punto, cabe remarcar que tanto el diseño como la implementación llevados a cabo para la parte algorítmica y gráfica del juego ESCOBA² han cumplido con los requisitos impuestos inicialmente. Además, se han implementado características adicionales que no estaban previstas inicialmente – *el sistema de retos, el soporte para pantallas de alta definición, etc.* – y se han llevado a cabo otras que no formaban parte de los requisitos pero que se barajaron como características deseables – *como el sistema de múltiples usuarios* -.

Además del satisfactorio desarrollo de la ESCOBA², es importante destacar una serie de hechos que no se traducen de forma directa en requisitos cumplidos pero que son tanto o más importantes: la calidad del diseño e implementación del producto.

Se destacan los siguientes puntos:

- Prácticamente todo el código se comparte entre las versiones de Windows y de Android.
- Hay una fuerte separación entre la implementación de la parte algorítmica y la implementación de la parte gráfica, siendo la primera totalmente agnóstica de la segunda.
- Gracias a la utilización de código nativo y un diseño e implementación que trata de ser lo más óptimo en la medida de lo posible, el rendimiento en terminales Android es muy bueno - incluso en terminales de entrada.
- Se soporta con excelente resolución gráfica tanto terminales móviles de baja resolución ($240\,x\,320$ píxeles) como tabletas con pantallas de alta definición (de hasta $2048x1024$ píxeles).
- Tanto el código que implementa el juego ESCOBA² como las librerías que se han utilizado en el desarrollo del mismo son completamente multiplataforma, por lo que un *port* a cualquier otro sistema operativo tradicional (*Mac OS X*, *Linux*,…) o móvil (*iOS* [17], *Tizen* [20],…) se reduce, principalmente, a establecer el entorno de desarrollo de código nativo para dicha plataforma y compilar el proyecto.
- El código fuente se encuentra dividido en 17 archivos *.cpp* y 19 archivos *.h*, de los cuales 5 archivos *.cpp* y 6 archivos *.h* pertenecen a la parte algorítmica mientras que el resto pertenecen a la parte gráfica.

9.3 Futuras líneas de trabajo

Finalmente y para concluir esta memoria, se tratarán una serie de futuras líneas de trabajo derivadas de este proyecto.

A continuación se listan y comentan las futuras líneas de trabajo que se han considerado más interesantes:

- **Añadir soporte multijugador online**: una característica muy interesante y que sin duda sería muy atractiva llevar a cabo sería la de incorporar la opción de jugar partidas multijugador contra otros jugadores humanos utilizando distintos dispositivos en tiempo real.

  Para ello, únicamente sería necesario modificar la parte gráfica, añadiendo toda la parte online por detrás e implementando las modificaciones necesarias en la interfaz. Al mismo tiempo, se podría seguir utilizando la parte de inteligencia artificial para jugar partidas entre varios jugadores humanos y 1 o 2 jugadores máquina (con un máximo de 4 jugadores en total).

  Al no ser una I.A. determinista (hay un componente aleatorio en función de la dificultad), bien habría que hacerla determinista (con números aleatorios pre-establecidos y conocidos en todos los terminales) o bien habría que desarrollar una arquitectura de tipo cliente(s)-servidor.

- **Añadir soporte para redes sociales**: una de las características que se encuentra en auge durante estos últimos años es la de socializar los juegos, integrando los sistemas de puntuación con las distintas redes sociales existentes, en aras de hacer los juegos más divertidos.

  Esta característica se podría añadir de forma paralela a la propuesta anteriormente, puesto que nos son mutuamente exclusivas.

- **Añadir soporte multilenguaje a la interfaz de usuario**: en la implementación actual de la ESCOBA² se utilizan muchas texturas con el texto ya integrado en la interfaz de usuario – *esto es, en la propia textura* -.

  Esto tiene la gran ventaja de ahorrar tiempo de ejecución puesto que no hemos de tener elementos adicionales para representar la gran mayoría de textos, en especial los textos estáticos; sin embargo, el gran problema es no tener soporte de múltiples idiomas, ya que sería necesario crear distintas versiones de la gran mayoría de texturas, una versión por idioma.

  Una elegante solución, aunque compleja de llevar a cabo, sería desacoplar el texto estático de las texturas y renderizarlo de forma dinámica a través de fuentes basadas en texturas, que se crean en tiempo de ejecución a partir de cadenas de caracteres. De esta forma podríamos tener soporte de múltiples idiomas sin apenas necesitar espacio de almacenamiento adicional.

- **Añadir efectos de sonido y partículas**: otra posible línea de trabajo sería la de añadir distintos efectos, tanto de sonido como gráficos, al juego.

  Este trabajo sería relativamente sencillo pues el motor *Cocos2D-x* [10] implementa soporte de sonidos y partículas.

  Además, se podría mejorar la interfaz de la vista de juego haciendo uso de estos efectos para notificar distintos eventos.

- **Añadir soporte de OpenGL 1.0** [5]: para concluir con las futuras líneas de trabajo propuestas, me gustaría mencionar la posibilidad de modificar el motor gráfico Cocos2D-x para añadir soporte de OpenGL 1.0 (presente en terminales *Android* [2] de primera generación).

  Versiones antiguas de este motor 2D estaban implementadas con OpenGL 1.1, por lo que se podría añadir compatibilidad hacia atrás y una vez tuviésemos una versión reciente de Cocos2D-x funcionando bajo OpenGL 1.1 se podría sustituir la funcionalidad de OpenGL 1.1 que no está presente en 1.0 por el equivalente disponible.

Como podemos ver, se han mencionado interesantes líneas de trabajo futuro que podrían mejorar y extender la versión actual del juego ESCOBA².

Para finalizar, me gustaría destacar que, como cualquier otro software, un juego está en constante desarrollo y siempre se puede mejorar y añadir soporte de nuevas características, plataformas y modos de juego, por lo que las posibles futuras líneas de trabajo son muchas y, recalco, muy interesantes.

10 Referencias

[1] Microsoft, «Microsoft Windows,» 2 Septiembre 2014. [En línea]. Available: http://windows.microsoft.com/es-es/windows/home. [Último acceso: 2 Septiembre 2014].

[2] Google, «Android,» 2 Septiembre 2014. [En línea]. Available: www.android.com. [Último acceso: 2 Septiembre 2014].

[3] ludoteka.com, «Escoba del 15,» 12 Diciembre 2013. [En línea]. Available: http://www.ludoteka.com/escoba.html. [Último acceso: 12 Diciembre 2013].

[4] «ARM Processors,» 2014. [En línea]. Available: http://www.arm.com/products/processors/index.php. [Último acceso: 3 Septiembre 2014].

[5] Khronos group, «OpenGL,» 2014. [En línea]. Available: http://www.opengl.org/. [Último acceso: 3 Septiembre 2014].

[6] UC3M, *Apuntes de Inteligencia en Redes de Ordenadores*, Madrid: UC3M, 2013.

[7] Rogue Wave Software, Inc, «CPU Cache miss,» 12 Diciembre 2013. [En línea]. Available: http://www.roguewave.com/portals/0/products/threadspotter/docs/2012.1/linux/manual_html/miss_ratio.html. [Último acceso: 12 Diciembre 2013].

[8] A. G. José González, «Recopilación de Técnicas de Predicción de Saltos,» 12 Diciembre 2013. [En línea]. Available: http://www.uv.es/~varnau/UPC-CEPBA-1996-11.pdf. [Último acceso: 12 Diciembre 2013].

[9] Oracle, «Java,» 2014. [En línea]. Available: https://www.java.com/es/. [Último acceso: 3 Septiembre 2014].

[10] Cocos2D-x, «Cocos2D-x.org,» 2013. [En línea]. Available: http://www.cocos2d-x.org/. [Último acceso: 10 Diciembre 2013].

[11] Ogre3D, «Ogre3D,» Ogre3D, 10 Diciembre 2013. [En línea]. Available: www.ogre3d.org. [Último acceso: 10 Diciembre 2013].

[12] J. Jouvie, «Bonzai Engine,» Bonzai Engine, 10 Diciembre 2013. [En línea]. Available: http://bonzaiengine.com/. [Último acceso: 10 Diciembre 2013].

[13] Unity Technologies, «Unity3D,» Unity3D, 10 Diciembre 2013. [En línea]. Available: www.unity3d.com. [Último acceso: 10 Diciembre 2013].

[14] Microsoft, «Visual Studio,» Microsoft, 10 Diciembre 2013. [En línea]. Available: http://www.visualstudio.com. [Último acceso: 10 Diciembre 2013].

[15] The Eclipse Foundation, «Eclipse,» 2014. [En línea]. Available: https://www.eclipse.org/. [Último acceso: 3 Septiembre 2014].

[16] Google, «Android NDK,» Google, 10 Diciembre 2013. [En línea]. Available: http://developer.android.com/tools/sdk/ndk/. [Último acceso: 10 Diciembre 2013].

[17] Apple, «Apple iOS,» 31 Agosto 2014. [En línea]. Available: https://www.apple.com/es/ios/what-is/. [Último acceso: 31 Agosto 2014].

[18] Blackberry, «Blackberry OS,» 2014. [En línea]. Available: http://es.blackberry.com/software/smartphones.html. [Último acceso: 3 Septiembre 2014].

[19] Microsoft, «Windows Phone,» 2014. [En línea]. Available: www.windowsphone.com. [Último acceso: 3 Septiembre 2014].

[20] Linux Foundation, «Tizen,» 2012. [En línea]. Available: https://www.tizen.org. [Último acceso: 3 Septiembre 2014].

[21] Cocos2D-x, «Cocos2D-x 2.2.1,» 1 Enero 2014. [En línea]. Available: http://cdn.cocos2d-x.org/cocos2d-x-3.0alpha1.zip. [Último acceso: 1 Enero 2014].

[22] Oracle, «JDK (Java Development Kit),» Oracle, 27 Abril 2014. [En línea]. Available: http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html. [Último acceso: 27 Abril 2014].

[23] Google, «Android SDK,» Google, 27 Abril 2014. [En línea]. Available: http://developer.android.com/sdk/index.html#ExistingIDE. [Último acceso: 27 Abril 2014].

[24] Cygwin, «Cygwin,» 27 Abril 2014. [En línea]. Available: http://www.cygwin.com/install.html. [Último acceso: 27 Abril 2014].

[25] Google, «Eclipse ADT (Android Development Tools) plugin,» 2014. [En línea]. Available: http://developer.android.com/tools/sdk/eclipse-adt.html. [Último acceso: 3 Septiembre 2014].

[26] Richard M. Stallman, «GNU/Linux,» 12 Abril 2014. [En línea]. Available: http://www.gnu.org/gnu/linux-and-gnu.es.html. [Último acceso: 3 Septiembre 2014].

[27] N. H. K. Patricio Salinas Caro, «Tutorial de UML,» [En línea]. Available: http://users.dcc.uchile.cl/~psalinas/uml/introduccion.html. [Último acceso: 3 Septiembre 2014].

[28] Google, «Android permissions - Write external storage,» Google, 4 Septiembre 2014. [En línea]. Available: http://developer.android.com/reference/android/Manifest.permission.html#WRITE_EXTERNAL_STORAGE. [Último acceso: 5 Septiembre 2014].

[29] cplusplus.com, «Function 'fopen',» 31 Agosto 2014. [En línea]. Available: http://www.cplusplus.com/reference/cstdio/fopen/. [Último acceso: 31 Agosto 2014].

[30] Google, «Context class reference, Android,» 31 Agosto 2014. [En línea]. Available: http://developer.android.com/reference/android/content/Context.html. [Último acceso: 31 Agosto 2014].

[31] Google, «JNI tips,» 31 Agosto 2014. [En línea]. Available: http://developer.android.com/training/articles/perf-jni.html. [Último acceso: 31 Agosto 2014].

[32] cocos2d-x, «Cocos2d-x CCDictionary,» 31 Agosto 2014. [En línea]. Available: www.cocos2d-x.org/wiki/CCDictionary. [Último acceso: 31 Agosto 2014].

[33] XML Core Working Group, «Extensible Markup Language (XML),» 17 Julio 2014. [En línea]. Available: www.w3.org/XML/. [Último acceso: 31 Agosto 2014].

[34] Apple, «NSDictionary reference - Apple,» 18 Septiembre 2013. [En línea]. Available: https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSDictionary_Class/Reference/Reference.html. [Último acceso: 31 Agosto 2014].

[35] W. Harris, «Aliasing and Filtering,» 4 Julio 2005. [En línea]. Available: http://www.bit-tech.net/hardware/2005/07/04/aliasing_filtering/1. [Último acceso: 29 Agosto 2014].

[36] Microsoft, «Bilinear filtering,» 28 Agosto 2014. [En línea]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/bb172357(v=vs.85).aspx. [Último acceso: 28 Agosto 2014].

[37] helpwithpcs, «Processor Basics - A basic guide to the processor,» 2014. [En línea]. Available: http://www.helpwithpcs.com/courses/processor-basics.htm. [Último acceso: 5 Septiembre 2014].

[38] «Native code vs Java comparison, PC & Android,» [En línea]. Available: https://code.google.com/p/android-benchmarks/. [Último acceso: 31 Agosto 2014].

[39] learnopengles, «A performance comparison between Java and C on the Nexus 5,» 8 Abril 2014. [En línea]. Available: http://www.learnopengles.com/a-performance-comparison-between-java-and-c-on-the-nexus-5/. [Último acceso: 31 Agosto 2014].

[40] Computer Science & Engineering - University of Washington, «In-order vs Out-of-order Execution,» 2006. [En línea]. Available: http://courses.cs.washington.edu/courses/csep548/06au/lectures/introOOO.pdf. [Último acceso: 31 Agosto 2014].

[41] IETF, «PNG (Portable Network Graphics) Specification,» IETF, 1 Marzo 1997. [En línea]. Available: http://tools.ietf.org/html/rfc2083. [Último acceso: 28 Agosto 2014].

[42] S. Dawson, «What is Dither?,» 21 Agosto 2003. [En línea]. Available: http://www.hifi-writer.com/he/dvdaudio/dither.htm. [Último acceso: 28 Agosto 2014].

[43] K. L. a. contributors, «pngquant,» 28 Agosto 2014. [En línea]. Available:

http://pngquant.org/. [Último acceso: 28 Agosto 2014].

[44] «PNGoo,» [En línea]. Available: https://code.google.com/p/pngoo/. [Último acceso: 28 Agosto 2014].

[45] Google, «Manifiest.permissions, Android,» 31 Agosto 2014. [En línea]. Available: http://developer.android.com/reference/android/Manifest.permission.html#WRITE_EXTERNAL_STORAGE. [Último acceso: 31 Agosto 2014].